Mondrian Wallet 2 First Flight Audit Report

Lead Auditors:

- Aditya Mishra

Table of Contents

Protocol Summary

The Mondrian Wallet team is back! And they decided "oh wow, zkSync has native account abstraction! Let's just use that. Also, we introduced a lot of bugs, so let's just make this codebase upgradeable, so that only the owner of the wallet can introduce functionality later as they see fit. Also, the NFT gimmick was silly so, let's not do that again."

If the contracts are upgradeable, we'll just be able to upgrade them if there is a bug, so no issues right?

To *really* understand this codebase, you'll want to learn about:

- Account Abstraction
- zkSync System Contracts
- Upgradable smart contracts via UUPS

Disclaimer

The Aditya Mishra team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	**Number of issues found**
High	2
Medium	2
Low	2
Info	3
Total	9

Findings

[]

Description:

Impact:

Proof of Concept:

Recommended Mitigation:

High

[H-1] Arbitrary Call Vulnerability

Description: This vulnerability which is shown in the `_executeTransaction` function that allows any transaction signer (including a compromised owner) to execute arbitrary calls to any address with any data and value. This can be exploited to drain the wallet's funds or interact with other contracts in unintended ways.

Impact:

1. An attacker gains access to the owner's private key through phishing or other means.
2. The attacker creates a transaction that transfers all funds to their address.
3. The attacker signs this transaction with the compromised key.
4. The attacker calls executeTransactionFromOutside with the malicious transaction.
5. The wallet executes the transaction, transferring all funds to the attacker.

Proof of Concept:

Recommended Mitigation:

Implement a whitelist of allowed addresses and function signatures that can be called. This could be done by adding a mapping of approved addresses and functions.

```
+    mapping(address => mapping(bytes4 => bool)) public approvedCalls;

+    function approveCall(address target, bytes4 functionSig) external onlyOwner {
+       approveCalls[target][functionSig] = true;
+    }

    function _executeTransaction(Transaction memory _transaction) internal {
        address to = address(uint160(_transaction.to));
        uint128 value = Utils.safeCastToU128(_transaction.value);
        bytes memory data = _transaction.data;
+        bytes4 functionSig = bytes4(data);

+        require(approvedCalls[to][functionSig], "Unapproved call");

        if (to == address(DEPLOYER_SYSTEM_CONTRACT)) {
            uint32 gas = Utils.safeCastToU32(gasleft());
            SystemContractsCaller.systemCallWithPropagatedRevert(gas, to, value,
```

```
data);
        } else {
            bool success;
            (success,) = to.call{value: value}(data);
            if (!success) {
                revert MondrianWallet2__ExecutionFailed();
            }
        }
    }
```

[H-2] Insecure external call

Description: The `MondrianWallet2` contract uses a low-level call in its `executeTransaction` function without properly handling the possibility of a failed execution. While the contract does revert with a custom error `MondrianWallet2__ExecutionFailed()`, it doesn't distinguish between different types of failures, which could lead to unexpected behavior and potential loss of funds.

Impact: If a transaction fails due to reasons other than running out of gas (e.g., the called contract reverts), the wallet contract will revert, but it won't provide any detailed information about the failure. This lack of granularity in error handling could make it difficult for users and integrating systems to understand why a transaction failed, potentially leading to repeated failed attempts or misinterpretation of the contract's state.

Proof of Concept:

Below is the test code which proves that there is a insecure external call done by the `executeTransaction` method.

Paste this code in the new test file and see the result

```
    // SPDX-License-Identifier: MIT
    pragma solidity ^0.8.24;

    import {Test, console} from "forge-std/Test.sol";
    import {MondrianWallet2} from "../src/MondrianWallet2.sol";
    import {Transaction} from "lib/foundry-era-contracts/src/system-
contracts/contracts/libraries/MemoryTransactionHelper.sol";
    import {BOOTLOADER_FORMAL_ADDRESS} from "lib/foundry-era-contracts/src/system-
contracts/contracts/Constants.sol";

    contract FailingContract {
        function fail() external pure {
            revert("Intentional failure");
        }
    }

    contract ProofOfConcept is Test {
        MondrianWallet2 wallet;
        FailingContract failingContract;
        address owner = address(0x1);

        function setUp() public {
```

```
            vm.prank(owner);
            wallet = new MondrianWallet2();
            failingContract = new FailingContract();
        }

        function testUncheckedLowLevelCall() public {
            Transaction memory transaction = Transaction({
                txType: 1,
                from: uint256(uint160(address(wallet))),
                to: uint256(uint160(address(failingContract))),
                gasLimit: 100000,
                gasPerPubdataByteLimit: 800,
                maxFeePerGas: 2000000000,
                maxPriorityFeePerGas: 1000000000,
                paymaster: 0,
                nonce: 0,
                value: 0,
                reserved: [uint256(0), uint256(0), uint256(0), uint256(0)],
                data: abi.encodeWithSignature("fail()"),
                signature:
hex"1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcd
ef1234567890abcdef1234567890abcdef1234567890abcdef00",
                factoryDeps: new bytes32[](0),
                paymasterInput: hex"",
                reservedDynamic: hex""
            });

            vm.prank(BOOTLOADER_FORMAL_ADDRESS);
            vm.expectRevert(bytes4(0x50058470)); // This is the selector for
MondrianWallet2__ExecutionFailed()
            wallet.executeTransaction(bytes32(0), bytes32(0), transaction);
        }
    }
```

Recommended Mitigation:

1. Use a try-catch block to handle the low-level call and capture more detailed error information:

```
    function executeTransaction(bytes32, bytes32, Transaction memory _transaction)
external payable requireFromBootLoaderOrOwner {
        try this.executeCall(_transaction.to, _transaction.value,
_transaction.data) {
            // Transaction succeeded
        } catch Error(string memory reason) {
            // The call reverted with a reason string
            revert MondrianWallet2__ExecutionFailed(reason);
        } catch (bytes memory lowLevelData) {
            // The call reverted without a reason string
            revert MondrianWallet2__ExecutionFailed("Unknown error");
        }
    }
```

```
    function executeCall(uint256 _to, uint256 _value, bytes memory _data) external
payable {
        (bool success, bytes memory result) = address(uint160(_to)).call{value:
_value}(_data);
        if (!success) {
            assembly {
                revert(add(result, 32), mload(result))
            }
        }
    }
```

2. Consider using OpenZeppelin's Address.functionCall or similar utilities that provide more robust error handling for low-level calls.
3. Implement a way to return detailed error information to the caller, possibly through events or returnable error codes.
4. Add extensive logging (events) for both successful and failed transactions to aid in debugging and monitoring.

Medium

[M-1] Centralization Risk

Description: The contract uses a single-owner model for the upgrades, creating a central point of failure. If the owner's key is compromised, an attacker could upgrade the contract to a malicious implementation. The contract uses a UUPS upgradeable pattern, but the _authorizeUpgrade function is empty, allowing the owner to upgrade the contract without any restrictions.

```
    contract MondrianWallet2 is IAccount, Initializable, OwnableUpgradeable,
UUPSUpgradeable {
```

and

```
    function _authorizeUpgrade(address newImplementation) internal override {}
```

Impact:

1. An attacker gains access to the owner's private key.
2. The attacker deploys a new malicious implementation contract.
3. The attacker calls the upgrade function to replace the current implementation with the malicious one.
4. All future interactions with the wallet now use the malicious code, potentially allowing the attacker to steal funds or perform other malicious actions.
5. The owner could maliciously upgrade the contract, potentially draining user funds or introducing backdoors.

Proof of Concept:

Recommended Mitigation:

Implement a timelock mechanism for upgrade:

```
    uint256 public constant UPGRADE_TIMELOCK = 2 days;
    address public pendingImplementation;
    uint256 public upgradeProposalTime;

    function proposeUpgrade(address newImplementation) external onlyOwner {
        pendingImplementation = newImplementation;
        upgradeProposalTime = block.timestamp;
        emit UpgradeProposed(newImplementation);
    }

    function _authorizeUpgrade(address newImplementation) internal override {
        require(newImplementation == pendingImplementation, "Invalid
implementation");
        require(block.timestamp >= upgradeProposalTime + UPGRADE_TIMELOCK,
"Timelock not expired");
        pendingImplementation = address(0);
        upgradeProposalTime = 0;
        emit UpgradeAuthorized(newImplementation);
    }

    event UpgradeProposed(address indexed newImplementation);
    event UpgradeAuthorized(address indexed newImplementation);
```

This solution will add a timelock period between proposing and executing an upgrade, giving users time to react if a malicious upgrade is proposed. It also improves transparency by emitting events for upgrade-related actions.

[M-2] Lack of Signature Replay Protection

Description: The contract doesn't implement protection against signature replay attacks. An attacker could intercept a valid transaction and replay it multiple times. We can see that in the function `_validateTransaction`

```
    bytes32 txHash = _transaction.encodeHash();
    address signer = ECDSA.recover(txHash, _transaction.signature);
    bool isValidSigner = signer == owner();
```

Impact:

1. Alice signs a transaction to send 1 ETH to Bob.
2. The transaction is broadcasted and executed successfully.
3. An attacker intercepts this transaction.
4. The attacker repeatedly calls `executeTransactionFromOutside` with the intercepted transaction.
5. The wallet executes the same transaction multiple times, potentially draining Alice's funds.

Proof of Concept:

Recommended Mitigation:

Implement a nonce-based replay protection:

```
    function _validateTransaction(Transaction memory _transaction) internal
returns (bytes4 magic) {
+        require(!usedNonces[_transaction.nonce], "Nonce already used");
+        usedNonces[_transaction.nonce] = true;

        SystemContractsCaller.systemCallWithPropagatedRevert(
            uint32(gasleft()),
            address(NONCE_HOLDER_SYSTEM_CONTRACT),
            0,
            abi.encodeCall(INonceHolder.incrementMinNonceIfEquals,
(_transaction.nonce))
        );

        // Check for fee to pay
        uint256 totalRequiredBalance = _transaction.totalRequiredBalance();
        if (totalRequiredBalance > address(this).balance) {
            revert MondrianWallet2__NotEnoughBalance();
        }

        // Check the signature
        bytes32 txHash = _transaction.encodeHash();
        address signer = ECDSA.recover(txHash, _transaction.signature);
        bool isValidSigner = signer == owner();
        if (isValidSigner) {
            magic = ACCOUNT_VALIDATION_SUCCESS_MAGIC;
        } else {
            magic = bytes4(0);
        }
        return magic;
}
```

Low

[L-1] Unused function should be removed

Description: The function `_authorizeUpgrade` which is not used in the whole codebase should be removed.

```
    function prepareForPaymaster(
```

```
    function _authorizeUpgrade(address newImplementation) internal override {}
```

Impact: If the function is not used in the codebase then it should be removed so that the contract does not require more gas to deploy the contract.

Recommended Mitigation: Remove the function.

[L-2] Lack of access control in `payForTransaction`

Description: The `payForTransaction` function lacks access control, allowing anyone to potentially pay for transactions on behalf of the wallet.

Impact:

1. An attacker notices a pending transaction from the wallet.
2. The attacker calls payForTransaction with a high gas price, front-running the original transaction.
3. This could lead to unexpected behavior or potential manipulation of transaction ordering.

Proof of Concept:

Recommended Mitigation:

Add access control to the function

```
    function payForTransaction(bytes32, bytes32, Transaction memory _transaction)
external payable onlyOwner {
        bool success = _transaction.payToTheBootloader();
        if (!success) {
            revert MondrianWallet2__FailedToPay();
        }
    }
```

Informational

[I-1] Different solidity versions are used in the contract.

Description: Use only one solidity version to compile all files.

Impact: Takes more time to compile the smart contract because of different versions.

[I-2] Unused import should be removed

Description: Unused import should not be present in the codebase, so that it does not make any complexities in auditing the contract.

```
    import {MessageHashUtils} from
"@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
```

Impact: Confusing the developer in writing the code and also auditor in identifying the vulnerabilities.

Recommended Mitigation: Better should be removed to avoid confusion to the developers.

[I-3] Missing input validation

Description: The function `validateTransaction` is missing input validation.

Impact: Lack of input validation could lead to unexpected behaviour if malicious transactions are submitted.

Proof of Concept:

Recommended Mitigation:

Add input validation for the transaction struct fields

```
    function _validateTransaction(Transaction memory _transaction) internal
returns (bytes4 magic) {
+        require(_transaction.to != address(0), "Invalid recipient");
+        require(_transaction.value <= address(this).balance, "Insufficient
balance");
        // rest of the function
    }
```