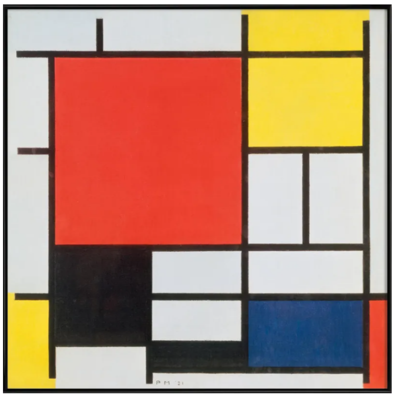

Mondrian Wallet v2

Version 1.0

Cyfrin.io

July 10, 2024

Codehawks First flight Mondrian Wallet v2

Seven Cedars

July 10, 2024

Prepared by: Seven Cedars

Table of Contents

Protocol Summary

The Mondrian Wallet team is back!

Contracts

MondrianWallet2.sol

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

This protocol was prepared for the Codehawk's firstflight program. It intentionally has numerous bugs and vulnerabilities.

Scope

- In Scope:

```
#-- interfaces
|   #-- 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.

Executive Summary

Severity	Number of Issues found
high	4
medium	4
low	1
total	9

Issues found

Findings

High

Missing `MondrianWallet2::receive` and `MondrianWallet2::fallback` functions make it impossible to move funds into the contract. Combined with the absence of a Paymaster account, it means it is impossible to validate transactions, breaking core functionality of the Account Abstraction.

Description: `MondrianWallet2.sol` is missing a receive and fallback function. It makes it impossible to move funds into the contract.

```
    constructor() {
        _disableInitializers();
    }

@>    // receive and / or fallback function should be here.

    function validateTransaction(bytes32, /*_txHash*/ bytes32, /*
        _suggestedSignedHash*/ Transaction memory _transaction)
```

Impact: Because `MondrianWallet2.sol` does not set up a paymaster account, the Account Abstraction will only work if `MondrianWallet2.sol` itself has sufficient `balance` to execute transactions. If not, the `MondrianWallet2::validateTransaction` will fail and return `bytes4(0)`
.

Lacking a receive and fallback function, it is impossible to move funds into the contract: Any empty call with ether will revert and calls to a function will return excess ether to the caller, leaving no funds in the contract.

This, in turn, means that the function `MondrianWallet2::_validateTransaction` will always revert:

```
        uint256 totalRequiredBalance = _transaction.
            totalRequiredBalance();
        // this conditional will always return false.
        if (totalRequiredBalance > address(this).balance) {
            revert MondrianWallet2__NotEnoughBalance();
        }
```

The only way to execute a transaction is by the owner of the contract through the `MondrianWallet2::executeTransaction`, which has the owner pay for the transaction directly. This approach of executing a transaction is exactly the same as the owner themselves executing the transaction directly, rendering the Account Abstraction meaningless.

An additional note on testing. This issue did not emerge in testing because the account is added ether through a cheat code in `MondrianWallet2Test.t.sol::setup`:

```
        vm.deal(address(mondrianWallet), AMOUNT);
```

Although common practice, it makes issues within funding contracts easy to miss.

Proof of Concept: 1. User deploys an account abstraction and transfers ownership to themselves. 2. User attempts to transfer funds to the contract and fails. 3. Bootloader attempts to validate transaction, fails. 4. User attempts to execute transaction directly through `MondrianWallet2::executeTransaction` and succeeds. In short, the only way transactions can be executed are directly by the owner of the contract, defeating the purpose of Account Abstraction.

Proof of Concept

First remove cheat code that adds funds to `mondrianWallet` account in `ModrianWallet2Test.t.sol::setup` [sic: note the missing n!].

```
- vm.deal(address(mondrianWallet), AMOUNT);
```

And set the proxy to payable:

```
- mondrianWallet = MondrianWallet2(address(proxy));
+ mondrianWallet = MondrianWallet2(payable(address(proxy)));
```

Then add the following to `ModrianWallet2Test.t.sol`.

```
      // Please note that you will also need --system-mode=true to run
          this test.
      function testMissingReceiveBreaksContract() public onlyZkSync {
        // setting up accounts
```

```
        uint256 AMOUNT_TO_SEND = type(uint128).max;
        address THIRD_PARTY_ACCOUNT = makeAddr("3rdParty");
        vm.deal(THIRD_PARTY_ACCOUNT, AMOUNT);
        // Check if mondrianWallet indeed has no balance.
        assertEq(address(mondrianWallet).balance, 0);

        // create transaction
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.
            mint.selector, address(mondrianWallet), AMOUNT);
        Transaction memory transaction = _createUnsignedTransaction(
            mondrianWallet.owner(), 113, dest, value, functionData);
        transaction = _signTransaction(transaction);

        // Act & assert
        // sending money directly to contract fails; it leaves contract
             with balance of 0.
        vm.prank(mondrianWallet.owner());
        (bool success, ) = address(mondrianWallet).call{value:
            AMOUNT_TO_SEND}("");
        assertEq(success, false);
        assertEq(address(mondrianWallet).balance, 0);

        // as a result, validating transaction by bootloader fails
        vm.prank(BOOTLOADER_FORMAL_ADDRESS);
        vm.expectRevert();
        mondrianWallet.validateTransaction(EMPTY_BYTES32, EMPTY_BYTES32
            , transaction);

        // the same goes for executeTransactionFromOutside
        vm.prank(THIRD_PARTY_ACCOUNT);
        vm.expectRevert();
        mondrianWallet.executeTransactionFromOutside(transaction);

        // also when eth is send with the transaction.
        vm.prank(THIRD_PARTY_ACCOUNT);
        vm.expectRevert();
        mondrianWallet.executeTransactionFromOutside{value:
            AMOUNT_TO_SEND}(transaction);

        // It is possible to execute function calls by owner through
            execute Transaction. But this defeats the purpose of Account
             Abstraction.
        vm.prank(mondrianWallet.owner());
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32,
             transaction);
        assertEq(usdc.balanceOf(address(mondrianWallet)), AMOUNT);
    }
```

Recommended Mitigation: Add a payable fallback function to the contract.

```
+   fallback() external payable {
// an additional check is needed so that the bootloader will never end
    up calling the fallback function.
+  assert(msg.sender != BOOTLOADER_FORMAL_ADDRESS);
+ }
```

Missing access control on `MondrianWallet2::_authorizeUpgrade` make it possible for anyone to call `MondrianWallet2::upgradeToAndCall` and permanently change its functionality.

Description: `MondrianWallet2` inherits `UUPSUpgradeable` from openZeppelin. This contract comes with a function `upgradeToAndCall` that upgrades a contract. It also comes with a requirement to include a `_authorizeUpgrade` function that manages access control. As noted in the `UUPSUpgradable` contract: > > The {_authorizeUpgrade} function must be overridden to include access restriction to the upgrade mechanism. >

However, the implementation of `_authorizeUpgrade` lacks any such access restrictions:

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

Impact: Because anyone can call `MondrianWallet2::upgradeToAndCall`, anyone can upgrade the contract to anything they want. First, this goes against the stated intention of the contract. From `README.md`: > > only the owner of the wallet can introduce functionality later > Second, it allows for a malicious user to disable the contract. Third, the the upgradeability can also be disabled (by having `_authorizeUpgrade` always revert), making it impossible to revert changes.

Proof of Concept: 1. A malicious user deploys an alternative `MondrianWallet2` implementation. 2. The malicious user calls `upgradeToAndCall` and sets the new address to their implementation. 3. The call does not revert. 4. From now on `MondrianWallet2` follows the functionality as set by the alternative `MondrianWallet2` implementation.

In the example below, all functions end up reverting - including the `upgradeToAndCall`. But any kind of change can be implemented, by any user, at any time.

Proof of Concept

Place the following code underneath the existing tests in `ModrianWallet2Test.t.sol`.

```
contract KilledImplementation is IAccount, Initializable,
    OwnableUpgradeable, UUPSUpgradeable  {
    error KilledImplementation__ContractIsDead();

    function initialize() public initializer {
```

```
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
    }

    function validateTransaction(bytes32, /*_txHash*/ bytes32, /*
       _suggestedSignedHash*/ Transaction memory _transaction)
        external
        payable
        returns (bytes4 magic)
    {
        if (_transaction.txType != 0) {
            revert KilledImplementation__ContractIsDead();
        }
        return bytes4(0);
    }

    function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
       _suggestedSignedHash*/ Transaction memory _transaction) external
        payable  {
        revert KilledImplementation__ContractIsDead();
    }

    function executeTransactionFromOutside(Transaction memory
       _transaction) external payable {
        revert KilledImplementation__ContractIsDead();
    }

    function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
       _suggestedSignedHash*/ Transaction memory _transaction) external
        payable {
        revert KilledImplementation__ContractIsDead();
    }

    function prepareForPaymaster(bytes32, /*_txHash*/ bytes32, /*
       _possibleSignedHash*/ Transaction memory /*_transaction*/)
       external payable {
        revert KilledImplementation__ContractIsDead();
    }

    function _authorizeUpgrade(address newImplementation) internal
       override {
        revert KilledImplementation__ContractIsDead();
    }
}
```

Then place the following among the existing tests:

```
    // Please note that you will also need --system-mode=true to run
       this test.
    function testAnyOneCanUpgradeAndKillAccount() public onlyZkSync {
        // setting up accounts
```

```
        address THIRD_PARTY_ACCOUNT = makeAddr("3rdParty");
        vm.deal(address(mondrianWallet), AMOUNT);
        // created an implementation (contract KilledImplementation below) in which every function reverts with the following error: `KilledImplementation__ContractIsDead`.
        KilledImplementation killedImplementation = new KilledImplementation();

        // create transaction
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.mint.selector, address(mondrianWallet), AMOUNT);
        Transaction memory transaction = _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData);
        transaction = _signTransaction(transaction);

        // Act
        // a random third party - anyone - can upgrade the wallet.
        // upgrade to `killedImplementation`.
        vm.expectEmit(true, false, false, false);
        emit Upgraded(address(killedImplementation));

        vm.prank(THIRD_PARTY_ACCOUNT);
        mondrianWallet.upgradeToAndCall(address(killedImplementation), "");

        // Assert
        // With the upgraded implementation, every function reverts with `KilledImplementation__ContractIsDead`.
        vm.prank(BOOTLOADER_FORMAL_ADDRESS);
        vm.expectRevert(KilledImplementation.KilledImplementation__ContractIsDead.selector);
        mondrianWallet.validateTransaction(EMPTY_BYTES32, EMPTY_BYTES32, transaction);

        vm.prank(mondrianWallet.owner());
        vm.expectRevert(KilledImplementation.KilledImplementation__ContractIsDead.selector);
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32, transaction);

        // crucially, also the upgrade call also reverts. Upgrading back to the original is impossible.
        vm.prank(mondrianWallet.owner());
        vm.expectRevert(KilledImplementation.KilledImplementation__ContractIsDead.selector);
        mondrianWallet.upgradeToAndCall(address(implementation), "");

        // ... and so on. The contract is dead.
    }
```

Recommended Mitigation: Add access restriction to `MondrianWallet2::_authorizeUpgrade`, for example the `onlyOwner` modifier that is part of the imported `OwnableUpgradable.sol` contract:

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

Missing validation check in `MondrianWallet2::executeTransactionFromOutside` allows anyone to execute transactions through the `MondrianWallet2` Account Abstraction. It breaks any kind of restriction of the contract and renders it unusable.

Description: The function `MondrianWallet2::executeTransactionFromOutside` misses a check on the result of `MondrianWallet2::_validateTransaction`. `_validateTransaction` does not revert when validation fails, but returns `bytes4(0)`. Without check, `MondrianWallet2::_executeTransaction` will always be called, even when validation of the transaction failed.

```
        function executeTransactionFromOutside(Transaction memory
            _transaction) external payable {
            _validateTransaction(_transaction);
            _executeTransaction(_transaction);
        }
```

Impact: Because of the missing check in `executeTransactionFromOutside`, anyone can sign and execute a transaction. It breaks any kind of restriction of the contract, allowing for immediate draining of all funds from the contract (among other actions) and renders it effectively unusable.

Proof of Concept: 1. A malicious user creates a transaction. 2. The malicious user signs the transaction with a random signature.
3. The malicious user calls `MondrianWallet2::executeTransactionFromOutside` with the randomly signed transaction. 4. The transaction does not revert and is successfully executed.

Proof of Concept

Place the following in `ModrianWallet2Test.t.sol`.

```
    // Please note that you will also need --system-mode=true to run
        this test.
    function
        testMissingValidateCheckAllowsExecutionUnvalidatedTransactions()
         public onlyZkSync {
        // setting up accounts
        address THIRD_PARTY_ACCOUNT = makeAddr("3rdParty");
```

```
        vm.deal(address(mondrianWallet), AMOUNT);
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.mint.selector, address(mondrianWallet), AMOUNT);
        Transaction memory transaction = _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData);

        // Act
        // we sign transaction with a random signature
        bytes32 unsignedTransactionHash = MemoryTransactionHelper.encodeHash(transaction);
        uint256 RANDOM_KEY = 0x00000000000000000000000000000000000000000000000000000000007ceda5;
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(RANDOM_KEY, unsignedTransactionHash);
        Transaction memory signedTransaction = transaction;
        signedTransaction.signature = abi.encodePacked(r, s, v);

        // and the transaction still passes.
        vm.prank(THIRD_PARTY_ACCOUNT);
        mondrianWallet.executeTransactionFromOutside(signedTransaction);
        assertEq(usdc.balanceOf(address(mondrianWallet)), AMOUNT);
    }
```

Recommended Mitigation: Add a check, using the result from `_validateTransaction`. Note that when validation succeeds, `_validateTransaction` returns the selector of `IAccount::validateTransaction`. `MondrianWallet2` already imports this value as `ACCOUNT_VALIDATION_SUCCESS_MAGIC`.

Add a check that `_validateTransaction` returns the value of `ACCOUNT_VALIDATION_SUCCESS_MAGIC`. If the check fails, revert with the error function `error MondrianWallet2__InvalidSignature()`. This error function is already present in `MondrianWallet2.sol`.

```
function executeTransactionFromOutside(Transaction memory _transaction) external payable {
+     bytes4 magic = _validateTransaction(_transaction);
-     _validateTransaction(_transaction);
+     if (magic != ACCOUNT_VALIDATION_SUCCESS_MAGIC) {
+          revert MondrianWallet2__InvalidSignature();
+    }
      _executeTransaction(_transaction);
    }
```

Missing Access Control on `MondrianWallet2::executeTransaction` allows for breaking of a fundamental invariant of ZKSync: the uniqueness of (sender, nonce) pairs in transactions.

Description: As the ZKSync documentation states: > > One of the important invariants of every blockchain is that each transaction has a unique hash. [...] > Even though these transactions would be technically valid by the rules of the blockchain, violating hash uniqueness would be very hard for indexers and other tools to process. [...] > One of the easiest ways to ensure that transaction hashes do not repeat is to have a pair (sender, nonce) always unique. > The following protocol [on ZKSync] is used: > - Before each transaction starts, the system queries the NonceHolder to check whether the provided nonce has already been used or not. > - If the nonce has not been used yet, the transaction validation is run. The provided nonce is expected to be marked as "used" during this time. > - After the validation, the system checks whether this nonce is now marked as used. >

In short, for ZKSync to work properly, each transaction that is executed needs to have a unique (sender, nonce) pair. The `MondrianWallet::validateTransaction` function ensures this invariance holds, by increasing the nonce with each validated transaction.

Usually, the bootloader of ZKSync will always call validate before executing a transaction and check if a transaction has already been executed. However, because in `MondrianWallet2` the owner of the contract can also execute a transaction, they can choose to execute a transaction multiple times - irrespective if this transaction has already been executed.

```
function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction)
```

Impact: As the owner of `MondrianWallet2` can execute a transaction multiple times, it breaks of a fundamental invariant of ZKSync: the uniqueness of (sender, nonce) pairs. It can potentially have serious consequences for the functioning of the contract.

Proof of Concept: 1. A user creates a transaction to mint usdc coins. 2. The user executes the transaction, with nonce 0. 3. The user executes the same transaction - again with nonce 0. 4. And again, and again.

Proof of Concept

Place the following in `ModrianWallet2Test.t.sol`.

```
    function testExecuteTransactionBreaksUniquenessNonce() public
        onlyZkSync {
        vm.deal(address(mondrianWallet), AMOUNT);
        uint256 amoundUsdc = 1e10;
        uint256 numberOfRuns = 3; // the number of times to execute a
            transaction.

```

```
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.
            mint.selector, address(mondrianWallet), amoundUsdc);
        Transaction memory transaction = _createUnsignedTransaction(
            mondrianWallet.owner(), 113, dest, value, functionData);
        transaction = _signTransaction(transaction);

        vm.startPrank(mondrianWallet.owner());
        for (uint256 i; i < numberOfRuns; i++) {
            // the nonce stays at 0.
            vm.assertEq(transaction.nonce, 0);
            // each time the execution passes without problem.
            mondrianWallet.executeTransaction(EMPTY_BYTES32,
                EMPTY_BYTES32, transaction);
        }
        vm.stopPrank();

        // this leaves the owner with 3 times the amount of usdc coins
            - because the contracts has been called three times. With
            the exact same sender-nonce pair.
        assertEq(usdc.balanceOf(address(mondrianWallet)), numberOfRuns
            * amoundUsdc);
    }
```

Recommended Mitigation: The simplest mitigation is to only allow the bootloader to call `executeTransaction`. This can be done by replacing the `requireFromBootLoaderOrOwner` modifier with the `requireFromBootLoader` modifier.

```
    function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
        _suggestedSignedHash*/ Transaction memory _transaction)
        external
        payable
+        requireFromBootLoader

    function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
        _suggestedSignedHash*/ Transaction memory _transaction)
        external
        payable
-        requireFromBootLoaderOrOwner
```

This also allows for the deletion of the `requireFromBootLoaderOrOwner` modifier in its entirety:

```
-    modifier requireFromBootLoaderOrOwner() {
-        if (msg.sender != BOOTLOADER_FORMAL_ADDRESS && msg.sender !=
    owner()) {
-            revert MondrianWallet2__NotFromBootLoaderOrOwner();
-        }
-        _;
```

```
-     }
```

Medium

When the owner calls the function `MondrianWallet2::renounceOwnership` any funds left in the contract are stuck forever.

Description: `MondrianWallet2` inherits the function `renounceOwnership` from openZeppelin's `OwnableUpgradeable`. This function simply transfers ownership to `address(0)`.

See `OwnableUpgradeable.sol`:

```
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }
```

As is noted in `OwnableUpgradeable.sol`: > > NOTE: Renouncing ownership will leave the contract without an owner, > thereby disabling any functionality that is only available to the owner. >

Making any kind of transaction depends on a signature of the owner. As such, no transactions are possible after the owner renounces their ownership. This includes transfer of funds out of the contract.

Impact: In the life cycle of an Abstracted Account, renouncing ownership is a likely final action. It is very easy to forget to transfer any remaining funds out of the contract before doing so, especially when doing so in an emergency. As such, it is quite likely that funds are left in the contract by accident.

Proof of Concept: 1. A user deploys `MondrianWallet2` and transfers ownership to their address. 2. The user transfers funds into the `MondrianWallet2` account. 3. The user renounces ownership and forgets to retrieve funds. 4. User's funds are now stuck in the account forever.

Proof of Concept

Place the following in `ModrianWallet2Test.t.sol`.

```
    // Please note that you will also need --system-mode=true to run this test.
    function testRenouncingOwnershipLeavesEthStuckInContract() public onlyZkSync {
        vm.deal(address(mondrianWallet), AMOUNT);
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.mint.selector, address(mondrianWallet), AMOUNT);
        Transaction memory transaction = _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData);
        transaction = _signTransaction(transaction);

```

```
        vm.prank(mondrianWallet.owner());
        mondrianWallet.renounceOwnership();

        vm.prank(ANVIL_DEFAULT_ACCOUNT);
        vm.expectRevert(MondrianWallet2.
            MondrianWallet2__NotFromBootLoaderOrOwner.selector);
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32,
             transaction);

        vm.prank(BOOTLOADER_FORMAL_ADDRESS);
        bytes4 magic = mondrianWallet.validateTransaction(EMPTY_BYTES32
            , EMPTY_BYTES32, transaction);
        vm.assertEq(magic, bytes4(0));
    }
```

Recommended Mitigation: One approach is to override `OwnableUpgradeable::renounceOwnership` function, adding a transfer of funds to the contract owner when `renounceOwnership` is called.

Note that it is probably best *not* to make renouncing ownership conditional on funds having been successfully transferred. In some cases it might be more important to immediately renounce ownership (for instance when keys of an account have been compromised) rather than retrieving all funds from the contract.

Add the following code to `MondrianWallet2.sol`:

```
+   function renounceOwnership() public override onlyOwner {
+      uint256 remainingFunds = address(this).balance;
+      owner().call{value: remainingFunds}("");
+      _transferOwnership(address(0));
+    }
```

`MondrianWallet2::payForTransaction` lacks access control, allowing a malicious actor to block a transaction by draining the contract prior to validation.

Description: According to the ZKsync documentation, the `payForTransaction` function is meant to be called only by the Bootloader to collect fees necessary to execute transactions.

However, because an access control is missing in `MondrianWallet2::payForTransaction` anyone can call the function. There is also no check on how often the function is called.

This allows a malicious actor to observe the transaction in the mempool and use its data to repeatedly call payForTransaction. It results in moving funds from `MondrianWallet2` to the ZKSync Bootloader.

```
@>   function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction)
```

```
        external
        payable
    {
```

Impact: When funds are moved from the `MondrianWallet2` to the ZKSync Bootloader, `MondrianWallet2::validateTransaction` will fail due to lack of funds. Also, when the bootloader itself eventually calls `payForTransaction` to retrieve funds, this function will fail.

In effect, the lack of access controls on `MondrianWallet2::payForTransaction` allows for any transaction to be blocked by a malicious user.

Please note thatthere is a refund of unused fees on ZKsync. As such, it is likely that `MondrianWallet2` will eventually receive a refund of its fees. However, it is likely a refund will only happen after the transaction has been declined.

Proof of Concept: Due to limits in the toolchain used (foundry) to test the ZKSync blockchain, it was not possible to obtain a fine grained understanding of how the bootloader goes through the life cycle of a 113 type transaction. It made it impossible to create a true Proof of Concept of this vulnerability. What follows is as close as possible approximation using foundry's standard test suite.

The sequence: 1. Normal user A creates a transaction. 2. Malicious user B observes the transaction. 3. Malicious user B calls `MondrianWallet2::payForTransaction` until `mondrianWallet2.balance < transaction.maxFeePerGas * transaction.gasLimit`. 4. The bootloader calls `MondrianWallet::validateTransaction`. 5. `MondrianWallet::validateTransaction` fails because of lack of funds.

Proof of Concept

Place the following in `ModrianWallet2Test.t.sol`.

```
    // You'll also need --system-mode=true to run this test
    function testBlockTransactionByPayingForTransaction() public
        onlyZkSync {
        // Prepare
        uint256 FUNDS_MONDRIAN_WALLET = 1e16;
        vm.deal(address(mondrianWallet), FUNDS_MONDRIAN_WALLET);
        address THIRD_PARTY_ACCOUNT = makeAddr("3rdParty");

        // create transaction
        address dest = address(usdc);
        uint256 value = 0;
        bytes memory functionData = abi.encodeWithSelector(ERC20Mock.
            mint.selector, address(mondrianWallet), AMOUNT);
        Transaction memory transaction = _createUnsignedTransaction(
            mondrianWallet.owner(), 113, dest, value, functionData);
        transaction = _signTransaction(transaction);

```

```
        // using information embedded in the Transaction struct, we can
           calculate how much fee will be paid for the transaction
        // and, crucially, how many runs we need to move sufficient
           funds from the Mondrian Wallet to the Bootloader until
           mondrianWallet2.balance < transaction.maxFeePerGas *
           transaction.gasLimit.
        uint256 feeAmountPerTransaction = transaction.maxFeePerGas *
            transaction.gasLimit;
        uint256 runsNeeded = FUNDS_MONDRIAN_WALLET /
            feeAmountPerTransaction;
        console2.log("runsNeeded to drain Mondrian Wallet:", runsNeeded
            );

        // Act
        // by calling payForTransaction a sufficient amount of times,
           the contract is drained.
        vm.startPrank(THIRD_PARTY_ACCOUNT);
        for (uint256 i; i < runsNeeded; i++) {
            mondrianWallet.payForTransaction(EMPTY_BYTES32,
                EMPTY_BYTES32, transaction);
        }
        vm.stopPrank();

        // Act & Assert
        // When the bootloader calls validateTransaction, it fails: Not
           Enough Balance.
        vm.prank(BOOTLOADER_FORMAL_ADDRESS);
        vm.expectRevert(MondrianWallet2.
            MondrianWallet2__NotEnoughBalance.selector);
        bytes4 magic = mondrianWallet.validateTransaction(EMPTY_BYTES32
            , EMPTY_BYTES32, transaction);
    }
```

Recommended Mitigation: Add an access control to the `MondrianWallet2::payForTransaction` function, allowing only the bootloader to call the function.

```
function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction)
        external
        payable
+       requireFromBootLoader
```

Missing checks on delegate calls allow for all public functions in `MondrianWallet2` to be called via a delegate call. This is not possible in traditional EoAs. It breaks the intended functionality of `MondrianWallet2` as described in its `README.md`.

Description: `MondrianWallet2:README.md` states that: > > The wallet should be able to do anything a normal EoA can do, … >

Because it is not a smart contract, a normal EoA cannot have functions that are called via a delegate call. However, all public functions in `MondrianWallet2` lack checks that disallow them to be called via a delegate call.

See the missing checks in the following functions:

```
function validateTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable requireFromBootLoader
```

```
function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable requireFromBootLoaderOrOwner
```

```
function executeTransactionFromOutside(Transaction memory
    _transaction) external payable
```

```
function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable
```

```
function prepareForPaymaster( bytes32, /*_txHash*/ bytes32, /*
    _possibleSignedHash*/ Transaction memory /*_transaction*/ )
    external payable
```

Impact: The lack of checks disallowing functions to be called via a delegate call, breaking the intended functionality of `MondrianWallet2`.

Recommended Mitigation: Create a modifier to check for delegate calls and apply this modifier to all public functions.

The mitigation below follows the example from `DefaulAccount.sol`, written by Matter Labs (creator of ZKSync).

```
+  modifier ignoreInDelegateCall() {
+      address codeAddress = SystemContractHelper.getCodeAddress();
+      if (codeAddress != address(this)) {
+          assembly {
+              return(0, 0)
+          }
+      }
+
+      _;
+ }
.
.
.
+  function validateTransaction(bytes32, /*_txHash*/ bytes32, /*
   _suggestedSignedHash*/ Transaction memory _transaction) external
```

```
    payable requireFromBootLoader ignoreInDelegateCall
-   function validateTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable requireFromBootLoader
.
.
.
+   function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable requireFromBootLoaderOrOwner ignoreInDelegateCall
-   function executeTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable requireFromBootLoaderOrOwner
.
.
.
+   function executeTransactionFromOutside(Transaction memory
    _transaction) external payable ignoreInDelegateCall
-   function executeTransactionFromOutside(Transaction memory
    _transaction) external payable
.
.
.
+   function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable ignoreInDelegateCall
-   function payForTransaction(bytes32, /*_txHash*/ bytes32, /*
    _suggestedSignedHash*/ Transaction memory _transaction) external
    payable
.
.
.
+   function prepareForPaymaster( bytes32, /*_txHash*/ bytes32, /*
    _possibleSignedHash*/ Transaction memory /*_transaction*/ ) external
     payable ignoreInDelegateCall
-   function prepareForPaymaster( bytes32, /*_txHash*/ bytes32, /*
    _possibleSignedHash*/ Transaction memory /*_transaction*/ ) external
     payable
```

Lacking control on return data at `MondrianWallet2::_executeTransaction` results in excessive gas usage, unexpected behaviour and unnecessary evm errors.

Description: The `_executeTransaction` function uses a standard `.call` function to execute the transaction. This function returns a `bool success` and `bytes memory data`.

However, ZKsync handles the return of this the `bytes data` differently than on Ethereum mainnet. In their own words, from the ZKsync documentation: > > 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 >

Even though the data field is not used (see the empty space after the comma in `(success,)` below), it does receive this data and build it up in memory *after* the call has succeeded.

```
(success,) = to.call{value: value}(data);
```

Impact: Some calls that ought to return a fail (due to excessive build up of memory) will pass the initial `success` check, and only fail afterwards through an `evm error`. Or, inversely, because `_executeTransaction` allows functions to return data and have it stored in memory, some functions fail that ought to succeed.

The above especially applies to transactions that call a function that returns large amount of bytes.

Additionally, - `_executeTransaction` is *very* gas inefficient due to this issue. - As the execution fails with a `evm error` instead of a correct `MondrianWallet2__ExecutionFailed` error message, functionality of frontend apps might be impacted.

Proof of Concept: 1. A contract has been deployed that returns a large amount of data. 2. `MondrianWallet2` calls this contract. 3. The contract fails with an `evm error` instead of `MondrianWallet2__ExecutionFailed`.

After mitigating this issue (see the Recommended Mitigation section below) 4. No call fail with an `evm error` anymore.

Proof of Concept

Place the following code after the existing tests in `ModrianWallet2Test.t.sol`:

```
contract TargetContract {
    uint256 public arrayStorage;

    constructor() {}

    function writeToArrayStorage(uint256 _value) external returns (
        uint256[] memory value) {
        arrayStorage = _value;

        uint256[] memory arr = new uint256[](_value);

        return arr;
    }
}
```

Place the following code in between the existing tests in `ModrianWallet2Test.t.sol`:

```
        // You'll also need --system-mode=true to run this test
    function testMemoryAndReturnData() public onlyZkSync {
        TargetContract targetContract = new TargetContract();
```

```
        vm.deal(address(mondrianWallet), 100);
        address dest = address(targetContract);
        uint256 value = 0;
        uint256 inputValue;

        // transaction 1
        inputValue = 310_000;
        bytes memory functionData1 = abi.encodeWithSelector(TargetContract.writeToArrayStorage.selector, inputValue, AMOUNT);
        Transaction memory transaction1 = _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData1);
        transaction1 = _signTransaction(transaction1);

        // transaction 2
        inputValue = 475_000;
        bytes memory functionData2 = abi.encodeWithSelector(TargetContract.writeToArrayStorage.selector, inputValue, AMOUNT);
        Transaction memory transaction2 = _createUnsignedTransaction(mondrianWallet.owner(), 113, dest, value, functionData2);
        transaction2 = _signTransaction(transaction2);

        vm.startPrank(ANVIL_DEFAULT_ACCOUNT);
        // the first transaction fails because of an EVM error.
        // this transaction will pass with the mitigations implemented (see above).
        vm.expectRevert();
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32, transaction1);

        // the second transaction fails because of an ExecutionFailed error.
        // this transaction will also not pass with the mitigations implemented (see above).
        vm.expectRevert();
        mondrianWallet.executeTransaction(EMPTY_BYTES32, EMPTY_BYTES32, transaction2);

        vm.stopPrank();
    }
```

Recommended Mitigation: By disallowing functions to write return data to memory, this problem can be avoided. In short, replace the standard `.call` with an (assembly) call that restricts the return data to length 0.

```
-    (success,) = to.call{value: value}(data);
+     assembly {
        success := call(gas(), to, value, add(data, 0x20), mload(data), 0, 0)
```

```
    }
```

Low

Including `MondrianWallet2::constructor` is unnecessary and can be left out to save gas.

Description: In normal EVM code, an upgradable contract needs to disable initialisers to avoid them writing data to storage. In ZkSync, because of the different way contracts are deployed, there is no difference between deployed code and constructor code. In more detail, from the ZKSync documentation: > > On Ethereum, the constructor is only part of the initCode that gets executed during the deployment of the contract and returns the deployment code of the contract. > On ZKsync, there is no separation between deployed code and constructor code. The constructor is always a part of the deployment code of the contract. > In order to protect it from being called, the compiler-generated contracts invoke constructor only if the isConstructor flag provided (it is only available for the system contracts). >

Impact: Disabling initializers in the constructor is unnecessary.

Recommended Mitigation: Remove the following code:

```
-    constructor() {
-        _disableInitializers();
-    }
```

False Positives