MondrianWallet2 Security Audit Report by MK (14th DEC 2024)

High Risk Findings

[H-1] Lack of access control in `MondrianWallet2::payForTransaction`

Description: The payForTransaction function in MondrianWallet2 lacks access control and frequency checks, allowing anyone to call it. A malicious actor can exploit this by observing transactions in the mempool and repeatedly calling the function, transferring funds from MondrianWallet2 to the ZKSync Bootloader.

Impact: The lack of access control on MondrianWallet2::payForTransaction allows malicious users to drain funds, causing MondrianWallet2::validateTransaction to fail due to insufficient balance. When the Bootloader later calls payForTransaction to collect fees, it will also fail. Although ZKSync refunds unused fees, the refund occurs only after the transaction is declined, further delaying recovery.

Proof of Concept:

1. By reading Line 101:

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

Recommended Mitigation:

1. Add Access Control to the function:

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

[H-2] Lack of access control in `_authorizeUpgrade`

Description: The `MondrianWallet2` contract inherits `UUPS` Upgradeable and overrides the `_authorizeUpgrade()` function, but lacks access control. This oversight allows anyone to set a malicious contract as the implementation, enabling an attacker to either `selfdestruct()` the proxy contract or exploit its logic.

Impact: Without access control on the `_authorizeUpgrade()` function, an attacker can upgrade the contract to a malicious implementation. This could allow them to exploit the contract's functionality, potentially leading to the destruction of the proxy contract (selfdestruct()) or other malicious activities, compromising the contract’s security and integrity.

Proof of Concept:

1. From Line: 167, it is observed.

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

Recommended Mitigation:

1. Add onlyOwner at the end.

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

[H-3] Missing `receive()` Function in `MondrianWallet2` Prevents Ether Reception

Description: The `receive()` function allows a smart contract wallet to accept direct Ether transfers by being triggered when funds are sent to the contract’s address. However, `MondrianWallet2.sol` lacks both the `receive()` and `fallback()` functions, preventing the contract from receiving Ether directly.

Impact: The absence of the `receive()` function in `MondrianWallet2.sol` prevents the contract from accepting direct Ether transfers, causing transactions to fail and potentially disrupting services that depend on receiving Ether directly.

Proof of Concept:

1. It is observed that there is no receive() or fallback() function in the codebase.

Recommended Mitigation:

1. Consider adding a `receive()` function to `MondrianWallet2.sol` contract.

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

[H-4] `executeTransactionFromOutside` Allows Anyone to Drain Wallet Funds

Description: The `MondrianWallet2` contract allows execution of signed transactions, but fails to properly validate the signature, allowing a malicious user to exploit this vulnerability. The `executeTransactionFromOutside` function calls `_validateTransactions`, but does not check the result, allowing transactions with invalid signatures to be executed. This can lead to unauthorized access, enabling attackers to drain funds or interact with other contracts.

Impact: A malicious user can self sign a transaction and drain all of the funds of the protocol. The impact is verified with the following test.

Proof of Concept:

1. It is observed from `Line: 96 to 99`.

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

2. Above function take input, pass it to `_validateTransaction` then call `_executeTransaction`. If we check `_validateTransaction` function given below -

```
function _validateTransaction(Transaction memory _transaction) internal returns
(bytes4 magic) {
        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;
    }
```

3. it is meant to check and verify the signature of the owner and returns `bytes4 magic` value accordingly. But this value is not being checked in `executeTransactionFromOutside` and `_executeTransaction` is being triggered. That will allow anybody to call functions on the behalf of owner and do whatever they want with user funds.

Recommended Mitigation:

1. Here is the recommendation to fix the code -

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

Medium Risk Findings

[M-1] Use `SignatureChecker` Over `ECDSA` for Signature Validation

Description:

The zkSync Era documentation recommends using `SignatureChecker` over `ECDSA` for signature validation due to the difference between ECDSA and contract signatures. While ECDSA signatures are fixed, contract signatures are revocable and can change over time, making them potentially valid at one block but invalid at the next. MondrianWallet currently expects ECDSA signatures, which makes it incompatible with zkSync accounts that use non-standard signing methods, as zkSync supports custom logic for signing transactions.

Impact: Accounts may utilize different signature schemes, making `ECDSA.recover` ineffective for accurate signature validation. This limitation could result in failed validations and potential security risks.

Proof of Concept:

1. It is observed in `Line:139`

```
    function _validateTransaction(Transaction memory _transaction) internal
returns (bytes4 magic) {
            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();
139            address signer = ECDSA.recover(txHash, _transaction.signature);
            bool isValidSigner = signer == owner();
            if (isValidSigner) {
                magic = ACCOUNT_VALIDATION_SUCCESS_MAGIC;
            } else {
                magic = bytes4(0);
            }
            return magic;
```

Recommended Mitigation:

1. Follow the recommendations in the ZkSync documentation:

- https://docs.zksync.io/build/quick-start/best-practices.html#gasperpubdatabyte-should-be-taken-into-account-in-development

Use zkSync Era's native account abstraction support for signature validation instead of this [ecrecover] function. We recommend not relying on the fact that an account has an ECDSA private key, since the account may be governed by multisig and use another signature scheme.

- https://docs.zksync.io/build/developer-reference/account-abstraction.html

The @openzeppelin/contracts/utils/cryptography/SignatureChecker.sol library provides a way to verify signatures for different account implementations. We strongly encourage you to use this library whenever you need to check that a signature of an account is correct

- https://docs.zksync.io/build/developer-reference/account-abstraction/building-smart-accounts

For smart wallets, we highly encourage the implementation of the EIP1271 signature-validation scheme. This standard is endorsed by the ZKsync team and is integral to our signature-verification library.

[M-2] Uncontrolled Return Data in MondrianWallet2::_executeTransaction Causes Excessive Gas and Errors

Description:

The `MondrianWallet2` will be deployed to ZKsync. In ZKsync `calls` have some differences. The codebase should account them and the calls should be implemented in the assembly language.

Proof of Concept:

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. Look at the following code.

```
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 {
+           success := call(gas(), to, value, add(data, 0x20), mload(data), 0, 0)
+       }
        if (!success) {
            revert MondrianWallet2__ExecutionFailed();
        }
    }
}
```

Low Risk Findings

[L-1] Failure to Follow Storage Gap Convention May Affect Future Contract Layouts

Description:

Storage gaps are used by a convention for reserving storage slots in a base contract. This allows future versions of that contract to use up those slots without affecting the storage layout of child contracts. The `__gap` variable is missing from the `MondrianWallet2` contract.

According to the covention it is useful to declare unused variables or the so-called storage gaps in base contracts that you may want to extend in the future, as a means of "reserving" those slots.

This can be done by adding the `__gap` unused variable to the beginning of teh contract, e.g. at Line 36 of MondrianWallet2.sol. The variable name `__gap` or a name starting with `__gap_` must be used for the array so that OpenZeppelin Upgrades will recognize the gap variables.

Impact

The missing `__gap` will affect the storage layout of future child contracts. This is due to an unfollowed convention for implementation of upgradeable contracts.

Recommendations

1. Add the `__gap` variable at the beginning of the `MondrianWallet2` contract. Look at the following code.

```
contract MondrianWallet2 is IAccount, Initializable, OwnableUpgradeable,
UUPSUpgradeable {
    using MemoryTransactionHelper for Transaction;

+   uint256[48] __gap;
+
    error MondrianWallet2__NotEnoughBalance();

    ...

}
```