Protocol Audit Report

Version 1.0

Ardeshir Gholami

May 12, 2024

Mondrain Wallet Audit Report

Ardeshir Gholami

12 Apr 2024

Prepared by: Ardeshir Lead Auditors: - 4rdii

Table of Contents

Protocol Summary

Users who create an account abstraction wallet with MondrianWallet will get a cool account abstraction wallet, with a random Mondrian art painting!

Disclaimer

I make 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
Likelihood	High	H	H/M	M
	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:

```
hash
```

Scope

```
./contracts/
___ MondrianWallet.sol
```

Roles

Executive Summary

Issues found

Severity	number of issues found
High	2
Medium	1
Low	1
Info	0
Total	3

Findings

High

[H-1] Anyone Can Use the Contract Because `MondrianWallet::_validateSignature` Does Not Check The Signer Address

Description: The `_validateSignature` internal function is used to check for signature validation of any UserOperations, but since it does not check the recoverd address of `ECDSA.recover` any UserOperation will pass this validation.

```
1   function _validateSignature(
2           PackedUserOperation calldata userOp,
3           bytes32 userOpHash
4       ) internal pure returns (uint256 validationData) {
5           bytes32 hash = MessageHashUtils.toEthSignedMessageHash(
                userOpHash);
6   @>      ECDSA.recover(hash, userOp.signature);
7           return SIG_VALIDATION_SUCCESS; // this always returns 0
8       }
```

Impact: There is no check wether the owner sent this operation or not and all signatures if recoverable will pass.

Proof of Concept:

Recommended Mitigation: Here is an example implementation from smartcontracts.tips guide to acoount abstraction, you can implement somthing similar to this:

```
function _validateSignature(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash
    ) internal pure returns (uint256 validationData) {
        bytes32 hash = MessageHashUtils.toEthSignedMessageHash(
           userOpHash);
-       ECDSA.recover(hash, userOp.signature);
+       address signer = ECDSA.recover(hash, userOp.signature);
+       if (signer != owner){
+           return SIG_VALIDATION_FAILED; // returns 1
+       }
        return SIG_VALIDATION_SUCCESS; // this always returns 0
    }
```

Recommended Mitigation: Just mod tokeId by 4 instead of 10:

```
  function tokenURI(
        uint256 tokenId
    ) public view override returns (string memory) {
        if (ownerOf(tokenId) == address(0)) {
            revert MondrainWallet__InvalidTokenId();
        }
-       uint256 modNumber = tokenId % 10;
+       uint256 modNumber = tokenId % 4;
        //@audit rarity is wrong 10% for 1 2 3 and 70% for 4
        if (modNumber == 0) {
            return ART_ONE;
        } else if (modNumber == 1) {
            return ART_TWO;
        } else if (modNumber == 2) {
            return ART_THREE;
        } else {
            return ART_FOUR;
        }
    }
```

[H-2] Wrong Implementation of ERC721

Description: The contract inherits ERC721 implmentation, but these account contracts are meant to be deployed once for every user cause the user should be owner of the contract. as a result of this every contract would be a ERC721 token with zero tokens in it. Also _mint function is never called so there will be no NFTs generated in the first place so if we add the `_mint` to constructor we also will have to somehow fix the many implementations problem.

```
@>  contract MondrianWallet is Ownable, ERC721, IAccount {
        .
        .
        .
        constructor(
            address entryPoint
        ) Ownable(msg.sender) ERC721("MondrianWallet", "MW") {
            i_entryPoint = IEntryPoint(entryPoint);
@>          _mint(owner(), 1); // this was added to fix zero nfts
    minted problem
        }
```

Impact: This will cause all Nfts having same URIs and also the nft collection will be seperated and meaningless.

Proof of Concept: if we add the `_mint` to `constructor` we can then use this test suit written using `hardhat-foundry` library. To prove the problem:

```
contract MWTest is Test {
    MondrianWallet mw;
    MockERC20 erc20;
    MockEntryPoint ep;
    address user = makeAddr("user");
    address secondUser = makeAddr("user2");
    uint256 constant MINT_AMOUNT = 10 ether;
    function setUp() public {
        if (block.chainid == 1) {
            //eth mainnet
            ep = MockEntryPoint(
                payable(address(0
                    x0000000071727De22E5E9d8BAf0edAc6f37da032)) //
                    entrypoint on mainnet
            );
        } else {
            ep = new MockEntryPoint();
        }
        vm.prank(user);
        mw = new MondrianWallet(address(ep));
        erc20 = new MockERC20();
        vm.deal(user, MINT_AMOUNT);
    }
    function testNFTImplementation() public {
        vm.prank(secondUser);
        MondrianWallet mw2 = new MondrianWallet(address(ep));
        address ownerOf2 = mw2.ownerOf(1);
        address ownerOf1 = mw.ownerOf(1);
        assertEq(ownerOf2, secondUser);
        assertEq(ownerOf1, user);
        assert(ownerOf2 != ownerOf1);
    }
}
```

Recommended Mitigation: To fix this we can use a factory contract to deploy the Wallet instances and have the factory be also the ERC721 implmentation, this way also there would not be an issue with minting cause we can mint the nft when the `createAccount` function is called. here is a simple factory from eth-infinitism github altered to have ERC721 in it.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.23;

import "@openzeppelin/contracts/utils/Create2.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ERC721} from "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "./SimpleAccount.sol"; // this would be the MondrianWallet.sol

/**
 * A sample factory contract for SimpleAccount
 * A UserOperations "initCode" holds the address of the factory, and a method call (to createAccount, in this sample factory).
 * The factory's createAccount returns the target account address even if it is already installed.
 * This way, the entryPoint.getSenderAddress() can be called either before or after the account is created.
 */
contract SimpleAccountFactory is ERC721 {
    SimpleAccount public immutable accountImplementation;

    constructor(IEntryPoint _entryPoint) {
        accountImplementation = new SimpleAccount(_entryPoint);
    }

    /**
     * create an account, and return its address.
     * returns the address even if the account is already deployed.
     * Note that during UserOperation execution, this method is called only if the account is not deployed.
     * This method returns an existing account address so that entryPoint.getSenderAddress() would work even after account creation
     */
    function createAccount(address owner,uint256 salt) public returns (SimpleAccount ret) {
        address addr = getAddress(owner, salt);
        uint256 codeSize = addr.code.length;
        if (codeSize > 0) {
            return SimpleAccount(payable(addr));
        }
        ret = SimpleAccount(payable(new ERC1967Proxy{salt : bytes32(salt)}(
                address(accountImplementation),
```

```
36                 abi.encodeCall(SimpleAccount.initialize, (owner))
37             )));
38         _mint(address(ret),1);
39     }
40
41     /**
42      * calculate the counterfactual address of this account as it would
            be returned by createAccount()
43      */
44     function getAddress(address owner,uint256 salt) public view returns
           (address) {
45         return Create2.computeAddress(bytes32(salt), keccak256(abi.
               encodePacked(
46             type(ERC1967Proxy).creationCode,
47             abi.encode(
48                 address(accountImplementation),
49                 abi.encodeCall(SimpleAccount.initialize, (owner))
50             )
51         )));
52     }
53 }
```

Medium

[M-1] Wrong Rarity of 4th URI in `MondrianWallet::tokenURI`

Description: The tokenURI function returns the URI of any given tokenId, but since there is only 4 URIs the tokenId should be moded by 4 not 10.

Impact: This will caues URIs 1 to 3 to be rare despite the protocol docs stating all have same rarity.

Proof of Concept:

```
1   function tokenURI(
2         uint256 tokenId
3     ) public view override returns (string memory) {
4         if (ownerOf(tokenId) == address(0)) {
5             revert MondrainWallet__InvalidTokenId();
6         }
7 @>      uint256 modNumber = tokenId % 10;
8         //@audit rarity is wrong 10% for 1 2 3 and 70% for 4
9         if (modNumber == 0) {
10            return ART_ONE;
11        } else if (modNumber == 1) {
12            return ART_TWO;
13        } else if (modNumber == 2) {
14            return ART_THREE;
15        } else {
16            return ART_FOUR;
```

```
17             }
18         }
```