MondrianWallet Security Audit Report by MK (16th DEC 2024)

High Risk Findings

[H-1] `_validateSignature` always return `SIG_VALIDATION_SUCCESS`, causing invalid signatures to be accepted

Description: The `_validateSignature` function is recovering the signature but does not check if it is correct, causing all signatures to be accepted.

Impact: Anybody can execute transactions without the owner's signature.

Proof of Concept: The signature check if supposed to check that the owner authorized the operation to be executed on its behalf. If any given signature is accepted, a malicious user can forge an operation and submit it for execution without the owner consent, potentially resulting in fund losses for the owner.

Recommended Mitigation: Use the return value of the recover function to return the correct `validationData`value.

```
    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_SUCCESS;
+        return SIG_VALIDATION_FAILED;
    }
```

[H-2] ZKSync does not have an `EntryPoint` contract, compromising some Mondriam wallet functionalities

Description: ZKSync network has its own native account abstraction implementation, which uses the Bootloader program as part of the account abstraction transaction process. This means there is no `EntryPoint` smart contract.

Impact: Some core account abstraction functionalities won't work, such as getting the account's balance or adding deposits via `EntryPoint`.

Proof of Concept: Because ZKSync network has its native account abstraction feature, there is no `EntryPoint` smart contract. Therefore, some functions from the `EntryPoint` used in `MondrianWallet` won't

work. These functions are:

```
* `MondrianWallet::getNonce`
* `MondrianWallet::getDeposit`
* `MondrianWallet::addDeposit`
```

On the other hand, if provided the correct Bootloader address as the constructor's `entryPoint` argument, `MondrianWallet::requireFromEntryPoint` and `MondrianWallet::requireFromEntryPointOrOwner` will work perfectly since `msg.sender` will be the Bootloader formal address, as stated in the official documentation (https://docs.zksync.io/zk-stack/components/zkEVM/bootloader.html).

Recommended Mitigation: An alternative would be to just not use those functionalities and do a direct transfer to the wallet when adding funds and read the wallet's address balance directly.

[H-3] No way for the user to mint NFT for the `MondrianWallet`

Description:

1. The `MondrianWallet` protocol mentions to mint a NFT to the one who creates account abstraction wallet but there is no way to mint NFT to the user.

2. But along with that, the `MondrianWallet` is actually the wallet for the user and has the ERC721 inherited and it is kind of irrelevant because NFT handling stuff should be handled via a MondrianWallet Deployer and the Deployer should allow the user to deploy their `MondrianWallet` and mint NFT.

Impact: Users can't have their NFT.

Proof of Concept:

- The vulnerability is present in the design of `MondrianWallet`, there is no way for the users to get their NFT.
- The NFT is associated with the `MondrianWallet` and there is no way for one to mint.
- As the wallets are associated with their own owners therefore it is irrelevant to associate ERC721 inheritance with the `MondrianWallet`.
- Instead there should be a deployer contract which should be a ERC721 contract and should allow users to create their `MondrianWallet` and mint their NFT.

Recommended Mitigation:

1. Create a Deployer contract that should inherit ERC721 contract and should allow the user to deploy their `MondrianWallet` along with the NFT.

2. Also, the ERC721 associated with `MondrianWallet` is insignificant as it is the wallet and not a source to mint NFT, therefore consider removing the ERC721 stuffs from `MondrianWallet`.

Medium Risk Findings

[M-1] Absence of automatic mint functionality

Description: The Mondrian Wallet smart contract, designed for NFT management and transactions, lacks an explicit call to the _safeMint() function in its constructor. This omission prevents the automatic minting of an NFT upon contract creation, a crucial feature for the wallet's intended functionality.

Impact: The failure to mint an NFT upon contract creation could significantly impact the utility and perceived value of the Mondrian Wallet. Users expecting to receive an NFT immediately upon interacting with the contract would be disappointed, potentially leading to decreased adoption and trust in the platform.

Proof of Concept: The primary vulnerability identified in the Mondrian Wallet contract is the absence of an explicit call to the _safeMint() function within its constructor. This oversight prevents the automatic minting of an NFT upon the contract's creation, which is a critical feature for the intended functionality of the wallet.

Recommended Mitigation: To address the identified issue and ensure the contract fulfills its intended functionality, the following recommendations are made:

Explicitly Call `_safeMint()` in the Constructor: Modify the contract's constructor to include a call to `_safeMint()`, ensuring an NFT is minted and assigned to the contract or a designated address upon deployment. This change will enable the automatic minting of an NFT upon contract creation, aligning with the wallet's intended design and functionality.

```
  constructor(address entryPoint) Ownable(msg.sender) ERC721("MondrianWallet",
"MW") {
        i_entryPoint = IEntryPoint(entryPoint);
        _safeMint(msg.sender, randomTokenId);
    }
```

[M-2] Uneven Distribution

Description: The `MondrianWallet::tokenURI` function in the contract returns different URIs based on the modulo of the token ID with a constant value. The implementation results in an uneven distribution, where one URI represents a significantly higher proportion of the results compared to the others.

Impact: The distribution is biased, with `ART_FOUR` having a much higher likelihood of being returned compared to the others.

Proof of Concept: Proof Of Code

You will need to add public mint function in MondrianWallet, I should note this mint function is only for proof purposes and should not be used as it's flawed

```
function mint(address to, uint256 _tokenIdCounter) public onlyOwner { _safeMint(to,
_tokenIdCounter); // Use _safeMint for ERC-721 compliance }
```

Place the following into `MondrianWallet.test.js`.

```
it("should have an even distribution of token URIs", async function () {
    const expectedURIs = [
```

```
            "ar://jMRC4pksxwYIgi6vIBsMKXh3Sq0dfFFghSEqrchd_nQ",
            "ar://8NI8_fZSi2JyiqSTkIBDVWRGmHCwqHT0qn4QwF9hnPU",
            "ar://AVwp_mWsxZO7yZ6Sf3nrsoJhVnJppN02-cbXbFpdOME",
            "ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM"
        ];
        const walletOwner = await mondrianWallet.owner()
        const distribution = [0, 0, 0, 0]; // To count occurrences of each URI
        for (let i = 0; i < 100; i++) { // Mint 100 tokens
            await mondrianWallet.mint(walletOwner,i); // You might need a mint
    function if it's not there
        }
        // Check token URIs for the first 100 tokens
        for (let i = 0; i < 100; i++) {
            const tokenURI = await mondrianWallet.tokenURI(i);
            const index = expectedURIs.indexOf(tokenURI);

            if (index >= 0) {
                distribution[index]++; // Count which URI is returned
            }
        }
        console.log(distribution)
        // [ 10, 10, 10, 70 ]
    });
```

Recommended Mitigation: with this each art has a 25% probability

```
    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;
            if (modNumber == 0) {
                return ART_ONE;
            } else if (modNumber == 1) {
                return ART_TWO;
            } else if (modNumber == 2) {
                return ART_THREE;
            } else {
                return ART_FOUR;
            }
        }
```

[M-3] `_validateSignature` always return `SIG_VALIDATION_SUCCESS`, causing invalid signatures to be accepted

Description: The `_validateSignature` function is recovering the signature but does not check if it is correct, causing all signatures to be accepted.

Impact: Anybody can execute transactions without the owner's signature.

Proof of Concept: The signature check if supposed to check that the owner authorized the operation to be executed on its behalf. If any given signature is accepted, a malicious user can forge an operation and submit it for execution without the owner consent, potentially resulting in fund losses for the owner.

Recommended Mitigation: Use the return value of the recover function to return the correct `validationData`value.

```
    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_SUCCESS;
+        return SIG_VALIDATION_FAILED;
    }
```

[M-4] Bad Distribution and Randomness

Description: The `MondrianWallet` contract suffers from a vulnerability related to the distribution and randomness of the non-fungible tokens (NFTs) it mints. The randomness is compromised because the `token ID` is used to assign the art, which is not random and allows users to predict which art they will receive based on the token ID. Additionally, the distribution is flawed because, despite having only four art options, the contract uses the `%10` operation, leading to an uneven distribution among the art options.

Impact: The impact of this vulnerability is significant in terms of user trust and the perceived value of the NFTs. The highly predictable and uneven distribution results in most users receiving the same art option (`ART_FOUR`), diminishing the rarity and desirability of the NFTs. This unfair assignment could lead to user dissatisfaction and a loss of trust in the protocol, as the NFTs minted do not reflect a fair or random allocation of art, which is essential for maintaining the integrity and appeal of the NFT collection.

Proof of Concept: To prove the vulnerability in the MondrianWallet contract regarding the poor distribution and randomness of NFTs, we first need to introduce a dummy `mint` function. This function, defined as mint, will allow minting of NFTs by incrementing a `token ID counter`. The function will look like this:

```
uint256 i;
function mint() external {
    _mint(msg.sender, i);
    i++;
}
```

Next, we write a test to `mint` a series of NFTs and verify the assigned art `URIs`. The test, written in JavaScript using the Hardhat framework, will demonstrate the predictable and uneven distribution of the NFTs. By

minting 10 tokens, we will be able to observe that the distribution is not random. Specifically, we can predict which art will be assigned to each token based on the token ID modulo 10. The code for the test is as follows:

```
it("Distribution and randomness is bad", async function () {
    let user1 = await ethers.getSigners();
    const numIterations = 10;
    for (let i = 0; i < numIterations; i++) {
        await mondrianWallet.mint();
    }

    // Assert the token URIs for the first 10 tokens
    assert.equal(await mondrianWallet.tokenURI(0),
"ar://jMRC4pksxwYIgi6vIBsMKXh3Sq0dfFFghSEqrchd_nQ");
    assert.equal(await mondrianWallet.tokenURI(1),
"ar://8NI8_fZSi2JyiqSTkIBDVWRGmHCwqHT0qn4QwF9hnPU");
    assert.equal(await mondrianWallet.tokenURI(2),
"ar://AVwp_mWsxZO7yZ6Sf3nrsoJhVnJppN02-cbXbFpdOME");
    assert.equal(await mondrianWallet.tokenURI(3),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(4),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(5),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(6),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(7),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(8),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
    assert.equal(await mondrianWallet.tokenURI(9),
"ar://n17SzjtRkcbHWzcPnm0UU6w1Af5N1p0LAcRUMNP-LiM");
});
```

The analysis of the test results demonstrates that the distribution of art among the NFTs is bad and not random. Based on the modulo arithmetic, tokens that have `IDs` ending with 3 through 9 are always assigned the same art URI (ART_FOUR), showing a 70% concentration. Meanwhile, the other art options are assigned only 30% of the time collectively (`ART_ONE`, `ART_TWO` and `ART_THREE` each 10%). This predictable and uneven distribution proves that the current implementation of the `tokenURI` function does not provide a fair or random assignment of art to the NFTs.

Recommended Mitigation: To solve the identified vulnerabilities, consider using `RANDOMIZER.AI`, which provides robust and secure randomness. This solution works on both `Ethereum` and `ZkSync`, ensuring a consistent and fair distribution of art assignments. By integrating `RANDOMIZER.AI`, the assignment of art pieces to tokens can be reliably randomized at the time of minting, improving the overall security and functionality of the NFT distribution process. To interact with `RANDOMIZER.AI`, we will add the following interface:

```
interface IRandomizer {
       function request(uint256 callbackGasLimit) external returns (uint256);
```

```
        function request(uint256 callbackGasLimit, uint256 confirmations) external
returns (uint256);
        function clientWithdrawTo(address _to, uint256 _amount) external;
    }
```

To improve the distribution and randomness of the assigned art the following code should be added to the contract. The `_assignArt` function should be called at the time of minting to assign art to the token, ensuring each token receives an art piece immediately upon creation. The assigned art should be stored in a mapping(uint256 => string) called `IdAndItsArt`, guaranteeing consistent and retrievable art for each token ID. The `tokenURI` function should be updated to retrieve the assigned art from this mapping, making it deterministic and ensuring it always returns the correct art.

```
constructor(address entryPoint, address randomizer) Ownable(msg.sender)
ERC721("MondrianWallet", "MW") {
        i_entryPoint = IEntryPoint(entryPoint);

        // The address for randomizer will depend on the network we want to work
on.
        i_randomizer = IRandomizer(randomizer);
    }

    IRandomizer private immutable i_randomizer;
    mapping(uint256=>string) public IdAndItsArt;
    uint256 random;

function _assignArt(uint256 tokenId) internal{
    // Request a random value from RANDOMIZER.AI
    i_randomizer.request(50000);

     // Based on the generated random value, assign an art piece to the tokenId.
     if (random == 0) {
            IdAndItsArt[tokenId]= ART_ONE;

        } else if (random == 1) {
            IdAndItsArt[tokenId]= ART_TWO;

        } else if (random == 2) {
            IdAndItsArt[tokenId]= ART_THREE;

        } else {
             IdAndItsArt[tokenId]= ART_FOUR;

        }
}

     function tokenURI(uint256 tokenId) public view override returns (string
memory) {
        if (ownerOf(tokenId) == address(0)) {
            revert MondrainWallet__InvalidTokenId();
        }
```

```
        // Return the art assigned to the tokenId.
        return  IdAndItsArt[tokenId];
    }

    function randomizerCallback(uint256 _id, bytes32 _value) external {
            //Callback can only be called by randomizer
            require(msg.sender == address(i_randomizer), "Caller not Randomizer");

            // Calculate the random value using modulo 4 to ensure fair
distribution
            random = uint256(_value) % 4;
        }

// Withdraw funds deposited with RANDOMIZER.AI
function randomizerWithdraw(uint256 amount) external onlyOwner {
            i_randomizer.clientWithdrawTo(msg.sender, amount);
        }
```

By generating a random value and performing an operation %4 on the random value, we solved the issue of bad randomness and uneven distribution of art among NFTs. This ensures a more secure, fair, and predictable assignment of art to each minted token.

[M-5] `_validateSignature` and `validateUserOp` do not follow official ERC4337 Standard

Description: `_validateSignature` do not follow official ERC4337 standard

Impact:

- Revert on signature mismatch, gonna cause reverting of whole batch. (As most of transactions are processed in batches in AA).
- Signature will be valid for infinite time can cause, signature replay attacks.
- Incomptability in Signature Aggregators

Proof of Concept: In `MondrianWallet` smartcontract, It's supposed be fully compatible with ERC4337. But it do not follow it strictly which can cause whole batch to revert. Which is not the intended behaviour by EIP4337.

```
function _validateSignature(PackedUserOperation calldata userOp, bytes32
userOpHash)
        internal
        pure
        returns (uint256 validationData)
    {
        bytes32 hash = MessageHashUtils.toEthSignedMessageHash(userOpHash);
@>       ECDSA.recover(hash, userOp.signature);
        return SIG_VALIDATION_SUCCESS;
    }
```

Here the function can revert in highlighted line if signature is invalid. As per official ERC4337, It shouldn't revert on invalid signature, rather return SIG_VALIDATION_FAILED.

In `validateUserOp` there is `uint256` return value, which is currently fixed to 0 if signature is valid else it's gonna revert. But as per official ERC4337 the return value for `validateUserOp` must be packed of `authorizer`, `validUntil` and `validAfter` timestamps.

- authorizer - 0 for valid signature, 1 to mark signature failure. Otherwise, an address of an authorizer contract. This ERC defines “signature aggregator” as authorizer.
- validUntil is 6-byte timestamp value, or zero for “infinite”. The UserOp is valid only up to this time.
- validAfter is 6-byte timestamp. The UserOp is valid only after this time.

Recommended Mitigation: Adhere to standard ERC4337 to avoid these bugs. These are some of the recomendations.

```
function _validateSignature(PackedUserOperation calldata userOp, bytes32
userOpHash)
        internal
        pure
        returns (uint256 validationData)
    {
        bytes32 hash = MessageHashUtils.toEthSignedMessageHash(userOpHash);
         address user =  ECDSA.recover(hash, userOp.signature);
         if(user != owner()) {
           return SIG_VALIDATION_FAILED;
         }
        // Calculate validUntil and validAfter timestamps
            uint48 validUntil = 0; // Set to zero for "indefinite" validity
            uint48 validAfter = 0; // No delay by default
            return uint256(user) << 208 | uint256(validUntil) << 160 |
uint256(validAfter) << 112;
        }
    }
```

[M-6] Contract can't receive NFTs sent with safeTransferFrom method

Description: Contract can't receive NFTs sent with `safeTransferFrom` method

Impact: This might lead to loss of value as NFTs sent wont be received.

Proof of Concept: The contract under consideration is designed to receive and store ERC721 tokens. However, certain smart wallets or contracts might utilize the `safeTransferFrom` method to send an NFT. The `safeTransferFrom` method checks for the implementation of the `onERC721Received` method when the recipient is a contract. This is to ensure that the recipient contract can appropriately handle ERC721 tokens.

Recommended Mitigation: Therefore, it's essential for the contract to extend the `ERC721Holder` contract from OpenZeppelin. The `ERC721Holder` contract has the `onERC721Received` method implemented, which allows the contract to correctly receive and store ERC721 tokens sent using `safeTransferFrom`. Do note that the current OZ implementation ERC721 includes a safeTransferFrom function.

[M-7] Accounts using non-standard signing methods won't work with `MondrianWallet`

Description: `MondrianWallet` expects ECDSA signatures, but ZkSync accounts might use non-standard signing methods. Any such accounts won't work with `MondrianWallet`.

Proof of Concept: zkSync's account abstraction allows accounts to use custom logic for signing transactions, not just ECDSA signatures. This means accounts using non-standard signing methods won't work with MondrianWallet as it currently relies on ECDSA.

Recommended Mitigation: Follow the recommendations in the ZkSync documentation:

1. 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.

1. 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

Low Risk Findings

[L-1] Invalid NFT URIS

Description: The URIs are currently formatted with a custom scheme (ar://), which may not be recognized or supported by the intended recipients or systems.

Impact: The impact of using invalid URIs includes:

Data Accessibility Issues: Recipients may not be able to access the data stored at these URIs, leading to incomplete or incorrect data retrieval. Compatibility Problems: The use of custom URL schemes may cause compatibility issues across different platforms and applications, limiting the reach and usability of the data.

Proof of Concept: The vulnerability arises from the use of custom URL schemes (ar://) for storing and accessing data. Custom URL schemes are not standardized across platforms and applications, leading to potential compatibility issues. This means that the URIs may not be correctly interpreted or handled by the intended recipients or systems, resulting in failed data retrieval or access.

Recommended Mitigation: To address the identified issue and ensure the URIs are valid and accessible, the following recommendation is made:

Transition to IPFS: Utilize IPFS for storing and accessing data. IPFS is a distributed file system that allows for permanent and decentralized storage of data. By storing the data on IPFS, you can ensure that it is accessible through a standard and widely supported protocol, improving compatibility and reliability. Storing Data on IPFS: Convert the data to be stored into a format compatible with IPFS (e.g., JSON, text files). Use the IPFS API

or CLI to add the data to IPFS, which will return a unique hash for the data. Accessing Data via IPFS: When retrieving the data, use the IPFS hash to fetch the data from the IPFS network. This ensures that the data can be accessed reliably, regardless of the recipient's platform or application. By implementing these recommendations, the smart contract can significantly improve the validity and accessibility of the URIs, ensuring that the data can be effectively stored and retrieved in a secure and reliable manner.