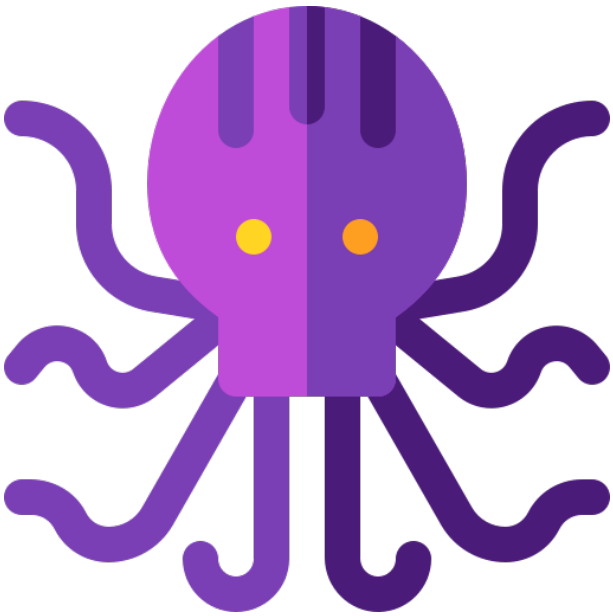

First Flight Mondrian Wallet Findings Report Report

Version 0.1

Yannick

May 22, 2024

First Flight Mondrian Wallet

Yannick

May 16, 2024

Prepared by: Yannick

Table of contents

Disclaimer

This code was created for Codehawks as the first flight. It is made with bugs and flaws on purpose.

Risk Classification

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

Audit Details

The findings described in this document correspond the following commit hash:

```
7874374
```

Scope

```
./contracts/
#-- MondrianWallet.sol
```

Contest Summary

Sponsor: First Flight #15

Dates: May 9th, 2024 - May 16th, 2024

See more contest details here

Results Summary

Number of findings

Severity	Number of issues found
High	1
Medium	2
Low	1
Info	0
Gas Optimizations	0
Total	4

Findings

High Risk Findings

[H-01] Missing check inside `_validateSignature` if the signature was signed by the owner

Relevant GitHub Links

Summary:

The absence of a check to verify that the owner of the contract is the signer of a transaction in the _validateSignature function introduces a significant vulnerability. This oversight allows any address to forge a signature that appears to come from the owner, potentially leading to unauthorized actions being performed on behalf of the owner.

Vulnerability Details:

The _validateSignature function is designed to validate the authenticity of a signature attached to a PackedUserOperation. It does so by hashing the userOpHash and then attempting to recover the signer's address from the signature using the ECDSA.recover function. However, the function does not perform any checks to ensure that the recovered address matches the expected owner of the contract. This means that an attacker could generate a signature that appears to be from the owner but is actually forged, allowing them to bypass the signature validation and potentially execute malicious actions.

Impact:

The primary impact of this vulnerability is the potential for unauthorized actions to be performed on the contract. An attacker who can forge a signature could manipulate the contract in ways that the owner did not intend, such as transferring funds, changing contract settings, or executing other actions

that require the owner's approval. This could lead to financial loss, loss of control over the contract, or other negative consequences for the owner and users of the contract.

Tools Used:

Hardhat

Recommendations:

Check if the recoveredAddress matches the owner address

```
function _validateSignature(PackedUserOperation calldata userOp,
    bytes32 userOpHash)
     internal
     returns (uint256 validationData)
{
     bytes32 hash = MessageHashUtils.toEthSignedMessageHash(userOpHash);
     address recoveredAddress = ECDSA.recover(hash, userOp.signature);
     if (recoveredAddress != owner()) {
      return SIG_VALIDATION_FAILED;
     }
     return SIG_VALIDATION_SUCCESS;
}
```

Medium Risk Findings

[M-01] Weak randomness for tokenURI

Relevant GitHub Links

Summary:

The protocol aims to mint a random NFT for new users when they create a new wallet, but in fact the protocol lacks in randomness

Vulnerability Details:

The vulnerability lies in the deterministic nature of the tokenURI function's logic. By using a modulus operation on the tokenId, the function maps every 10th token to the same URI (ART_ONE), every second token to another URI (ART_TWO), and so on. This predictable pattern reduces the randomness of the output, potentially leading to vulnerabilities such as predictability of token URIs

Impact:

Predictability: Attackers could predict the URI of tokens based on their IDs, which could lead to unauthorized access or manipulation of token metadata.

Tools Used:

Hardhat

Recommendations

To mitigate the identified vulnerability and enhance the randomness of the tokenURI function, consider the following recommendation:

Implement something like Chainlink VRF to achive randomness.

[M-02] Absence of automatic mint functionality

Relevant GitHub Links

Summary:

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.

Vulnerability Details:

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.

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.

Tools Used:

Hardhat

Recommendations:

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.

```
1    constructor(address entryPoint) Ownable(msg.sender) ERC721("
        MondrianWallet", "MW") {
2           i_entryPoint = IEntryPoint(entryPoint);
3           _safeMint(msg.sender, randomTokenId);
4       }
```

Low Risk Findings

[L-01] Invalid NFT URIS

Relevant GitHub Links

Summary:

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

Vulnerability Details:

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.

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.

Tools Used:

Hardhat

Recommendations:

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.