Security Review Report
NM-0112 ETHERSPOT WALLET CONTRACTS

(Aug 15, 2023)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 System Overview 4
- 4.1 wallet/Proxy.sol . . . 4
- 4.2 wallet/EtherspotWalletFactory.sol . . . 4
- 4.3 wallet/EtherspotWallet.sol & interfaces/IEtherspotWallet.sol . . . 5
- 4.4 access/AccessController.sol . . . 6
- 4.5 paymaster/Whitelist.sol & interfaces/IWhitelist.sol . . . 7
- 4.6 paymaster/EtherspotPaymaster.sol & interfaces/IEtherspotPaymaster.sol . . . 8
- 4.7 paymaster/BasePaymaster.sol . . . 9
- 4.8 helpers/UniversalSignatureValidator.sol . . . 9
- 4.9 interfaces/IERC721Wallet.sol . . . 9

5 Risk Rating Methodology 10

6 Issues 11
- 6.1 [Medium] Guardians can cosign an old proposal . . . 11
- 6.2 [Medium] Any guardian can remove owners from the Etherspot wallet . . . 12
- 6.3 [Medium] Improve the proposals' voting design . . . 13
- 6.4 [Low] In case a post user operation reverts, _postOp(...) function totally refunds the sponsor . . . 14
- 6.5 [Low] proposalId 0 can be cosigned by guardians without being proposed . . . 14
- 6.6 [Info] Owners cannot remove themselves from the wallet . . . 15
- 6.7 [Info] Proposals history/traceability is not accessible from the smart contract storage . . . 15
- 6.8 [Info] Removed guardian's signature in the latest proposal remains valid . . . 16
- 6.9 [Info] Resolved proposals can be discarded afterward . . . 16
- 6.10 [Info] Unnecessary inheritance in Whitelist.sol . . . 16
- 6.11 [Info] Unused variables in EtherspotWallet.sol . . . 17
- 6.12 [Info] EntryPoint address should be immutable in EtherspotWallet contract . . . 17
- 6.13 [Info] EtherspotWallet can be initialized with an invalid EntryPoint contract address . . . 18
- 6.14 [Info] QuorumNotReached event has an incorrect parameter name . . . 18
- 6.15 [Best Practice] onlyOwner modifier name represents incorrect logic . . . 18
- 6.16 [Best Practices] Apply Checks-Effects-Interactions pattern . . . 19
- 6.17 [Best Practices] Contract Whitelist does not use IWhitelist.sol . . . 19
- 6.18 [Best Practices] Functions that can have external visibility . . . 19
- 6.19 [Best Practices] Interface and implementation are not matching in EtherspotWallet.sol . . . 20
- 6.20 [Best Practices] Remove unused UniversalSignatureValidator contract from the codebase . . . 20
- 6.21 [Best Practices] Unclear function names in the Whitelist Contract . . . 21
- 6.22 [Best Practices] Unnecessary usage of input parameter . . . 21
- 6.23 [Best Practices] Unused WhitelistInitialized event in Whitelist.sol . . . 21
- 6.24 [Best Practices] NewOwnerProposal struct is not reordered to consume less gas . . . 22
- 6.25 [Best Practices] accountImplementation cannot be updated . . . 22
- 6.26 [Best Practices] checkSponsorFunds name is confusing . . . 23

7 Etherspot Wallet upgrade architecture 24
- 7.1 Current EtherspotWallet upgrade architecture . . . 24
- 7.2 Suggested upgrade architecture . . . 25

8 Documentation Evaluation 26

9 Test Suite Evaluation 27
- 9.1 Contracts Compilation . . . 27
- 9.2 Tests Output . . . 27
- 9.3 Code Coverage . . . 29
- 9.4 Slither . . . 29

10 About Nethermind 30

1 Executive Summary

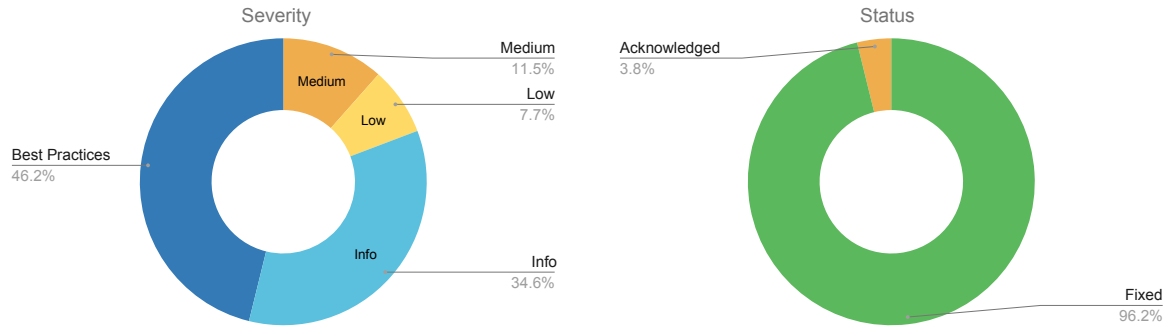
This document presents the security review performed by Nethermind on the EtherSpot Wallet Smart Contracts. The smart contracts implement the functionalities of the EtherSpot wallet. `EtherspotWallet` is an `EIP-4337` compliant smart contract serving as a multi-ownership wallet. It enables multiple owners to control a single account and execute transactions through an `EntryPoint` contract. It also integrates with the `EtherspotPaymaster` contract, allowing sponsors to cover gas costs for user operations.

The wallet implements an access control mechanism by inheriting the `AccessController` contract. This mechanism enables the wallet to have multiple owners and guardians. Guardians play a crucial role in account recovery; they participate in a voting mechanism to authorize adding new owners to the wallet. To reach a quorum and validate an owner, at least 60% of guardians must validate a vote, and a minimum of three guardians is required to initiate the voting process. To simplify the creation of new wallets, `EtherspotWalletFactory` comes into play, where users can effortlessly generate new wallets by calling the `createAccount` function.

`EtherspotPaymaster` is the smart contract that allows external signers to endorse `UserOperations` and take responsibility for covering the associated gas costs. The paymaster and the wallet engage in a dual-signature mechanism: the paymaster signs to commit to paying for gas, while the wallet signs to verify its identity and ownership of the account. Additionally, this contract incorporates a Whitelist functionality by inheriting a `Whitelist` contract, where sponsors should add specific users they wish to cover gas expenses for.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** 27 points of attention, where 3 are classified as `Medium`, 2 are classified as `Low`, and 21 are classified as `Best Practices` or `Informational`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the architecture for upgrading the contracts. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (3), **Low** (2), **Undetermined** (0), **Informational** (9), **Best Practices** (12). **Distribution of status: Fixed** (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (27), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug. 1, 2023
Response from Client	Aug. 10, 2023
Final Report	Aug. 15, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/etherspot/etherspot-prime-contracts/blob/6379fb0f6a0f538151bd6003f69fc05cc32fabe5
Commit Hash (Audit)	6379fb0f6a0f538151bd6003f69fc05cc32fabe5
Documentation	docs folder
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/wallet/EtherspotWallet.sol	126	30	23.8%	24	180
2	src/wallet/EtherspotWalletFactory.sol	87	27	31.0%	13	127
3	src/wallet/Proxy.sol	24	9	37.5%	3	36
4	src/helpers/UniversalSignatureValidator.sol	107	13	12.1%	12	132
5	src/access/AccessController.sol	190	3	1.6%	26	219
6	src/paymaster/Whitelist.sol	73	5	6.8%	13	91
7	src/paymaster/EtherspotPaymaster.sol	147	35	23.8%	25	207
8	src/paymaster/BasePaymaster.sol	55	40	72.7%	13	108
9	src/interfaces/IERC721Wallet.sol	7	1	14.3%	1	9
10	src/interfaces/IEtherspotPaymaster.sol	32	53	165.6%	8	93
11	src/interfaces/IEtherspotWallet.sol	25	1	4.0%	12	38
12	src/interfaces/IWhitelist.sol	25	1	4.0%	6	32
	Total	**898**	**218**	**24.3%**	**156**	**1272**

3 Summary of Issues

	Finding	Severity	Update
1	Guardians can cosign an old proposal	Medium	Fixed
2	Any guardian can remove owners from the `Etherspot wallet`	Medium	Fixed
3	Improve the proposals' voting design	Medium	Fixed
4	In case a post user operation reverts, `_postOp(...)` function totally refunds the sponsor	Low	Fixed
5	`proposalId` 0 can be cosigned by guardians without being proposed	Low	Fixed
6	Owners cannot remove themselves from the wallet	Info	Fixed
7	Proposals history/traceability is not accessible from the smart contract storage	Info	Fixed
8	Removed guardian's signature in the latest proposal remains valid	Info	Acknowledged
9	Resolved proposals can be discarded afterward	Info	Fixed
10	Unnecessary inheritance in `Whitelist.sol`	Info	Fixed
11	Unused variables in `EtherspotWallet.sol`	Info	Fixed
12	`EntryPoint` address should be immutable in `EtherspotWallet` contract	Info	Fixed
13	`EtherspotWallet` can be initialized with an invalid `EntryPoint` contract address	Info	Fixed
14	`QuorumNotReached` event has an incorrect parameter name	Info	Fixed
15	`onlyOwner` modifier name represents incorrect logic	Best Practices	Fixed
16	Apply Checks-Effects-Interactions pattern	Best Practices	Fixed
17	Contract `Whitelist` does not use `IWhitelist.sol`	Best Practices	Fixed
18	Functions that can have `external` visibility	Best Practices	Fixed
19	Interface and implementation are not matching in `EtherspotWallet.sol`	Best Practices	Fixed
20	Remove unused `UniversalSignatureValidator` contract from the codebase	Best Practices	Fixed
21	Unclear function names in the `Whitelist` Contract	Best Practices	Fixed
22	Unnecessary usage of input parameter	Best Practices	Fixed
23	Unused `WhitelistInitialized` event in `Whitelist.sol`	Best Practices	Fixed
24	`NewOwnerProposal` struct is not reordered to consume less gas	Best Practices	Fixed
25	`accountImplementation` cannot be updated	Best Practices	Fixed
26	`checkSponsorFunds` name is confusing	Best Practices	Fixed

4 System Overview

The audit covers the following smart contracts and interfaces:

- wallet/Proxy.sol
- wallet/EtherspotWalletFactory.sol
- wallet/EtherspotWallet.sol & interfaces/IEtherspotWallet.sol
- access/AccessController.sol
- paymaster/Whitelist.sol & interfaces/IWhitelist.sol
- paymaster/EtherspotPaymaster.sol & interfaces/IEtherspotPaymaster.sol
- paymaster/BasePaymaster.sol
- helpers/UniversalSignatureValidator.sol
- interfaces/IERC721Wallet.sol

4.1 wallet/Proxy.sol

This contract presents a standard proxy implementation similar to the one provided by OpenZeppelin in this link. It is used as a proxy for the newly created wallet contracts. The contract code is shown below.

```
contract Proxy {
    bytes32 internal constant _IMPLEMENTATION_SLOT = 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;

    constructor(address _singleton) {
        require(_singleton != address(0), "Invalid address provided");
        assembly {
            sstore(_IMPLEMENTATION_SLOT, _singleton)
        }
    }

    fallback() external payable {
        address target;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            target := sload(_IMPLEMENTATION_SLOT)
            calldatacopy(0, 0, calldatasize())
            let success := delegatecall(gas(), target, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            if eq(success, 0) {
                revert(0, returndatasize())
            }
            return(0, returndatasize())
        }
    }
}
```

4.2 wallet/EtherspotWalletFactory.sol

This contract is a factory for creating new wallet accounts using a proxy pattern. It allows for the efficient deployment of multiple wallet contracts using the same implementation address.

The contract emits the following event when a new wallet is created.

```
event AccountCreation(
    address indexed wallet,
    address indexed owner,
    uint256 index
);
```

It has one immutable storage variable `accountImplementation` that stores the wallet implementation address.

```
address public immutable accountImplementation;
```

The contract has the following public/external methods:

1. `constructor()`: The contract constructor function, it deploys a new `EtherspotWallet` contract and initializes the `accountImplementation` variable with its address. This address is immutable and will serve as the implementation of the proxy wallets.

2. `accountCreationCode()`: This function allows users to retreive the creation code for deploying a proxy contract, which can be used to calculate the predicted address of a deployed proxy.

3. `createAccount(IEntryPoint entryPoint, address owner, uint256 index)`: The main function of this contract, it enables users to create a new wallet by passing the `entryPoint` address, the wallet `owner` address, and an `index` as input parameters. The function deploys a proxy contract pointing to the wallet implementation stored in `accountImplementation` variable. It returns the account address.

4. `getAddress(IEntryPoint entryPoint, address owner, uint256 index)`: This function calculates and returns the address of a wallet based on the `entryPoint`, `owner`, and `index` values, as it would be returned by `createAccount(...)`.

4.3 wallet/EtherspotWallet.sol & interfaces/IEtherspotWallet.sol

The `EtherspotWallet` contract is an upgradeable multi-ownership wallet implementation. It provides functionalities that enable users to manage funds, execute function calls, manage owners and guardians for account recovery, and verify signatures. The contract allows owners to interact with an entry point contract, which is the main interface for executing user operations. It also includes mechanisms for executing batch function calls, enabling efficient execution of multiple operations within a single transaction. The interface `IEtherspotWallet.sol` defines the different functions and events to be implemented by the `EtherspotWallet`, the code is presented below.

```
import {IEntryPoint} from "../../account-abstraction/contracts/interfaces/IEntryPoint.sol";

interface IEtherspotWallet {
    event EtherspotWalletInitialized(
        IEntryPoint indexed entryPoint,
        address indexed owner
    );
    event EtherspotWalletReceived(address indexed from, uint256 indexed amount);
    event EntryPointChanged(address oldEntryPoint, address newEntryPoint);
    event OwnerAdded(address newOwner, uint256 blockFrom);
    event OwnerRemoved(address removedOwner, uint256 blockFrom);

    function nonce() external view returns (uint256);
    function entryPoint() external view returns (IEntryPoint);
    receive() external payable;
    function execute(address dest, uint256 value, bytes calldata func) external;
    function executeBatch(address[] calldata dest, bytes[] calldata func) external;
    function getDeposit() external view returns (uint256);
    function addDeposit() external payable;
    function isOwner(address _owner) external view returns (bool);
    function isGuardian(address _guardian) external view returns (bool);
    function updateEntryPoint(address _newEntryPoint) external;
}
```

The contract defines the following events.

```
event EtherspotWalletInitialized(IEntryPoint indexed entryPoint, address indexed owner);
event EtherspotWalletReceived(address indexed from, uint256 indexed amount);
event EntryPointChanged(address oldEntryPoint, address newEntryPoint);
```

We summarize below the external and public functions implemented by the `IEtherspotWallet.sol` contract:

1. `execute(address dest, uint256 value, bytes calldata func) external onlyOwnerOrEntryPoint(address(entryPoint()))`: Main function used to execute a transaction. Only the owner or the entry point contract can call this method.

2. `executeBatch(address[] calldata dest, uint256[] calldata value, bytes[] calldata func)`: This method allows the owner or the entry point contract to execute multiple transactions in a batch. It takes arrays of inputs that should have the same length.

3. `updateEntryPoint(address _newEntryPoint) external onlyOwner`: This method updates the entry point contract associated with the wallet to a new address. Only the owner can call this method.

4. `isValidSignature(bytes32 hash, bytes calldata signature) external view returns (bytes4 magicValue)`: This method implements the `isValidSignature` function from the `ISignatureValidator` interface. It verifies the validity of a signature based on a provided `hash` and `signature` data. If the signature is valid and the signer is the owner of the wallet, it returns the `ERC1271_SUCCESS` magic value; otherwise, it returns `bytes4(0xffffffff)`.

5. `receive() external payable`: Fallback function triggered when the contract receives Ether. It emits an event indicating the sender address and amount of Ether received.

6. `entryPoint() public view virtual override returns (IEntryPoint)`: This method returns the current entry point contract address associated with the wallet.

7. `getDeposit() public view returns (uint256)`: Returns the current deposit balance of the wallet in the entry point contract.

8. `initialize(IEntryPoint anEntryPoint, address anOwner) public virtual initializer`: This method initializes the wallet by setting the `anEntryPoint` address and adding `anOwner` as the owner of the wallet. It can be called only once after deployment.

9. `addDeposit() public payable`: This method allows users to deposit funds to the `EntryPoint` contract via the wallet.

10. `withdrawDepositTo(address payable withdrawAddress, uint256 amount) public onlyOwner`: This method allows a wallet owner to withdraw funds from the wallet's deposit to a specified `withdrawAddress`. The `amount` parameter indicates the amount to be withdrawn.

4.4 access/AccessController.sol

The `AccessController` contract provides a mechanism for managing access control within `EtherspotWallet.sol`, enabling multiple owners and guardians. Owners are allowed to add or remove owners and guardians in the wallet, while guardians can propose and approve new owners. The contract ensures a quorum is reached before adding a new owner by requiring 60% of guardian approvals.

The contract defines the following events.

```
event OwnerAdded(address newOwner);
event OwnerRemoved(address removedOwner);
event GuardianAdded(address newGuardian);
event GuardianRemoved(address removedGuardian);
event ProposalSubmitted(
    uint256 proposalId,
    address newOwnerProposed,
    address proposer
);
event QuorumNotReached(
    uint256 proposalId,
    address newOwnerProposed,
    uint256 guardiansApproved
);
event ProposalDiscarded(uint256 proposalId);
```

The contract defines the struct `NewOwnerProposal`.

```
struct NewOwnerProposal {
    address newOwnerProposed;
    uint256 approvalCount;
    address[] guardiansApproved;
    bool resolved;
}
```

The contract has the following storage variables.

```
uint128 immutable MULTIPLY_FACTOR = 1000;
uint16 immutable SIXTY_PERCENT = 600;
uint256 public ownerCount;
uint256 public guardianCount;
uint256 public proposalId;
mapping(address => bool) private owners;
mapping(address => bool) private guardians;
mapping(uint256 => NewOwnerProposal) private proposals;
```

It offers multiple functions to check and manage owners, guardians and proposals within the wallet. We summarize these functions below:

1. `function isOwner(address _address) public view returns (bool)`: Function that checks if the specified address is an owner.

2. `function isGuardian(address _address) public view returns (bool)`: Function that checks if the specified address is a guardian.

3. `function addOwner(address _newOwner) external onlyOwner`: Adds a new owner to the wallet. Only an owner can call this function.

4. `function removeOwner(address _owner) external onlyOwnerOrGuardian`: Removes an existing owner, requires the wallet to have at least two owners. This function can be called by a wallet owner or a guardian.

5. `function addGuardian(address _newGuardian) external onlyOwner`: Adds a new guardian to the wallet. It requires the caller to be an owner.

6. `function removeGuardian(address _guardian) external onlyOwner`: Removes an existing guardian from the wallet. It requires the caller to be an owner.

7. `function getProposal(uint256 _proposalId) public view returns (address ownerProposed_, uint256 approvalCount_, address[] memory guardiansApproved_, bool resolved_)` : This function returns the details of a proposal identified by the given `_proposalId`. It returns the proposed owner's address, the approval count, the array of guardians who approved the proposal, and a boolean indicating if the proposal is resolved.

8. `function discardCurrentProposal() external onlyOwnerOrGuardian`: This external function allows the owner or guardian to discard the current proposal. By discarding it, the proposal is marked as resolved without adding the owner to the wallet.

9. `function guardianPropose(address _newOwner) external onlyGuardian`: This function is used by guardians to propose a new owner. It requires the wallet to have at least 3 guardians, creates a new proposal with the proposed owner and the guardian's approval.

10. `function guardianCosign(uint256 _proposalId) external onlyGuardian`: This function allows a guardian to cosign a proposal. If quorum is reached (60% of guardians cosigned the proposal), the proposal is marked as resolved and the proposed owner is added to the wallet.

4.5 paymaster/Whitelist.sol & interfaces/IWhitelist.sol

The main goal of the `Whitelist` contract is to provide a mechanism for managing a whitelist of accounts associated with a specific `EtherspotPaymaster` contract. It enables sponsors to add or remove accounts they want to pay gas for, from the whitelist. The contract provides an external function to check if an account is whitelisted for a specific sponsor. Additionally, it emits events to track the initialization of the whitelist, the addition and removal of accounts, and batch operations on the whitelist. The interface `interfaces/IWhitelist.sol` defines the different functions and events to be implemented by the `Whitelist` contract. The code is presented below.

```
interface IWhitelist {
    event WhitelistInitialized(address owner, string version);
    event AddedToWhitelist(address indexed paymaster, address indexed account);
    event AddedBatchToWhitelist(
        address indexed paymaster,
        address[] indexed accounts
    );
    event RemovedFromWhitelist(
        address indexed paymaster,
        address indexed account
    );
    event RemovedBatchFromWhitelist(
        address indexed paymaster,
        address[] indexed accounts
    );

    function check(
        address _sponsor,
        address _account
    ) external view returns (bool);

    function add(address _account) external;

    function addBatch(address[] calldata _accounts) external;

    function remove(address _account) external;

    function removeBatch(address[] calldata _accounts) external;
}
```

The contract defines the following events.

```
event WhitelistInitialized(address owner);
event AddedToWhitelist(address indexed paymaster, address indexed account);
event AddedBatchToWhitelist(
    address indexed paymaster,
    address[] indexed accounts
);
event RemovedFromWhitelist(
    address indexed paymaster,
    address indexed account
);
event RemovedBatchFromWhitelist(
    address indexed paymaster,
    address[] indexed accounts
);
```

The contract has one storage variable `whitelist`, storing the whitelisted accounts for each sponsor address.

```
mapping(address => mapping(address => bool)) private whitelist;
```

`Whitelist` contract defines the following public and external functions:

1. `check(address _sponsor, address _account) external view returns (bool)`: This function checks whether a given account is whitelisted for a specific sponsor.

2. `add(address _account) external` : This function allows anyone to add an account to the whitelist for the calling entity (`msg.sender`). It emits the `AddedToWhitelist` event.

3. `addBatch(address[] calldata _accounts) external`: This function allows batch addition of multiple accounts to the whitelist. It emits the `AddedBatchToWhitelist` event upon successful batch addition.

4. `remove(address _account) external`: This function allows the removal of an account from the whitelist of the caller. It emits the `RemovedFromWhitelist` event upon successful removal.

5. `removeBatch(address[] calldata _accounts) external`: This function allows batch removal of multiple accounts from the caller whitelist. It emits the `RemovedBatchFromWhitelist` event upon successful batch removal.

4.6 paymaster/EtherspotPaymaster.sol & interfaces/IEtherspotPaymaster.sol

`EtherspotPaymaster` contract is a smart contract that allows an external signer to sign a given UserOperation and pay for the gas costs of executing it. The contract leverages the functionality provided by the `BasePaymaster` contract and the `Whitelist` contract to achieve its objectives. One of the key features of the `EtherspotPaymaster` contract is its trust in an external signer. The external signer performs off-chain verification of the user operation and signs the transaction accordingly. The paymaster relies on this signature to agree to pay for the gas costs associated with the transaction. `EtherspotPaymaster` contract also incorporates the `Whitelist` functionality. By inheriting from the `Whitelist` contract, sponsors funds are used to pay for gas of accounts they whitelisted.

The interface `interfaces/IEtherspotPaymaster.sol` is presented below.

```
import {IEntryPoint} from "../../account-abstraction/contracts/interfaces/IEntryPoint.sol";

interface IEtherspotWallet {
    event EtherspotWalletInitialized(
        IEntryPoint indexed entryPoint,
        address indexed owner
    );
    event EtherspotWalletReceived(address indexed from, uint256 indexed amount);
    event EntryPointChanged(address oldEntryPoint, address newEntryPoint);
    event OwnerAdded(address newOwner, uint256 blockFrom);
    event OwnerRemoved(address removedOwner, uint256 blockFrom);

    function nonce() external view returns (uint256);
    function entryPoint() external view returns (IEntryPoint);
    receive() external payable;
    function execute(address dest, uint256 value, bytes calldata func) external;
    function executeBatch(address[] calldata dest, bytes[] calldata func) external;
    function getDeposit() external view returns (uint256);
    function addDeposit() external payable;
    function isOwner(address _owner) external view returns (bool);
    function isGuardian(address _guardian) external view returns (bool);
    function updateEntryPoint(address _newEntryPoint) external;
}
```

The contract provides functions to manage sponsors funds, check if enough funds are deposited, verifying the signatures and handling the post operation process. We summarize below the main public and external functions of the contract:

- `function depositFunds() external payable nonReentrant`: The contract allows users to deposit funds by calling the this function. The deposited funds are credited to the sender's account and stored in the `sponsorFunds` mapping. This mapping keeps track of the funds deposited by each sponsor.

- `function withdrawFunds(address payable _sponsor, uint256 _amount) external nonReentrant`: Users can withdraw their deposited funds by calling this function. The contract verifies that the caller is the rightful sponsor and that they have sufficient funds deposited.

- `function checkSponsorFunds(address _sponsor) public view returns (uint256)`: The contract provides this function to allow users to check the amount of funds they have deposited in the paymaster.

- `function getHash(UserOperation calldata userOp, uint48 validUntil, uint48 validAfter) public view returns (bytes32)`: This function generates the hash used for off-chain signing and on-chain validation. This hash covers all fields of the `UserOperation`, except the `paymasterAndData`.

Additionally, the contract overrides two functions from the `BasePaymaster` contract:

- `function _validatePaymasterUserOp(UserOperation calldata userOp, bytes32 /*userOpHash*/, uint256 requiredPreFund) internal override returns (bytes memory context, uint256 validationData)`: This function verifies the validity of the user operation and the signature provided by the external signer. This validation includes checking the paymaster's whitelist, verifying the sponsor's deposited funds, and debiting the required pre-fund amount.

- `function _postOp(PostOpMode mode, bytes calldata context, uint256 actualGasCost) internal override`: This function handles the post-operation logic based on the execution mode and actual gas cost. It ensures that any unused funds are credited back to the sponsor's account.

The contract defines the following events.

```
event WhitelistInitialized(address owner);
event AddedToWhitelist(address indexed paymaster, address indexed account);
event AddedBatchToWhitelist(
    address indexed paymaster,
    address[] indexed accounts
);
event RemovedFromWhitelist(
    address indexed paymaster,
    address indexed account
);
event RemovedBatchFromWhitelist(
    address indexed paymaster,
    address[] indexed accounts
);
```

The contract has the following storage variable.

```
IEntryPoint private _entryPoint;
```

4.7 paymaster/BasePaymaster.sol

BasePaymaster is an abstract contract that serves as a base implementation for a paymaster contract. The contract is based on eth-infinitism account-abstraction `BasePaymaster` contract, removing the two functionalities `deposit()` and `withdrawTo()` which were moved to the wallet contract.

4.8 helpers/UniversalSignatureValidator.sol

The contract is a universal signature validator that provides functionality to verify signatures in Ethereum transactions. This implementation is taken from the reference implementation prposed in are the reference implementation of EIP-6492 about `Signature Validation for Predeploy Contracts`.

4.9 interfaces/IERC721Wallet.sol

The interface defines the function `isValidSignature(...)`.

```
interface IERC1271Wallet {
    function isValidSignature(
        bytes32 hash,
        bytes calldata signature
    ) external view returns (bytes4 magicValue);
}
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Guardians can cosign an old proposal

File(s): `AccessController.sol`

Description: The `guardianCosign(...)` function in the `AccessController` contract allows guardians to co-sign a proposal. However, an inconsistency in the usage of proposal ID can lead to guardians co-signing old proposals that were discarded.

The function uses both the `_proposalId` input argument and the `proposalId` state variable in its logic. The `_proposalId` parameter is checked to be a valid ID. However, when checking whether a proposal is not resolved, and that quorum is reached, the function uses the `proposalId` state variable. This inconsistency can lead to unintended behavior. A guardian can co-sign a proposal that was discarded previously, adding the discarded proposal's owner to the wallet and marking the latest proposal as resolved.

```
function guardianCosign(uint256 _proposalId) external onlyGuardian {
    require(_proposalId <= proposalId, "ACL:: invalid proposal id");
    require(
        !_checkIfSigned(_proposalId),
        "ACL:: guardian already signed proposal"
    );
    ////////////////////////////////////////////////////////////////////////////////
    // @audit: Checking if latest proposal is resolved, but _proposalId can be different
    ////////////////////////////////////////////////////////////////////////////////
    require(
        !proposals[proposalId].resolved,
        "ACL:: proposal already resolved"
    );
    proposals[_proposalId].guardiansApproved.push(msg.sender);
    proposals[_proposalId].approvalCount += 1;
    address newOwner = proposals[_proposalId].newOwnerProposed;

```

```
    ////////////////////////////////////////////////////////////////////////
    // @audit: Resolving latest proposal based on quorum check of _proposalId
    ////////////////////////////////////////////////////////////////////////
    if (_checkQuorumReached(_proposalId)) {
        proposals[proposalId].resolved = true;
        _addOwner(newOwner);
    } else {
        emit QuorumNotReached(
            _proposalId,
            newOwner,
            proposals[_proposalId].approvalCount
        );
    }
}
```

Recommendation(s): Remove the `_proposalId` parameter and exclusively use the `proposalId` state variable. Since the system is designed to have only one active proposal at a time (with ID stored in `proposalId`), there is no need for the parameter.

Status: Fixed

Update from the client: Recommended fix applied. Can be found at the following commit https://github.com/etherspot/etherspot-prime-contracts/commit/67791238b9025785c3a98051a24cb55a65f312b6.

6.2 [Medium] Any guardian can remove owners from the `Etherspot wallet`

File(s): src/access/AccessController.sol

Description: The `removeOwner()` function from the `AccessController` contract removes an `owner` from the wallet. This function has the `onlyOwnerOrGuardian` modifier, allowing any owner or guardian to call it. A malicious `guardian` can remove `owners` from the `Etherspot wallet`. The code is shown below.

```
modifier onlyOwnerOrGuardian() {
    require(
        isOwner(msg.sender) ||
            msg.sender == address(this) ||
            isGuardian(msg.sender),
        "ACL:: only owner or guardian"
    );
    _;
}
////////////////////////////////////////////////////////////////////////////////
// @audit Guardian should not be able to delete an owner
////////////////////////////////////////////////////////////////////////////////
function removeOwner(address _owner) external onlyOwnerOrGuardian {
    _removeOwner(_owner);
}
```

Recommendation(s): Consider restricting this action to only wallet owners.

Status: Fixed

Update from the client: `removeOwner()` now uses `onlyOwner` modifier.

6.3 [Medium] Improve the proposals' voting design

File(s): src/access/AccessController.sol

Description: Below, we describe two issues with the voting system design.

Proposal design does not account for disagree votes: The current implementation of the proposal's vote doesn't work when the consensus discards a proposal. That is because the contract doesn't keep track of the guardians who disagree, and there is no way to vote against a proposal.

As an example, if there are 3 guardians, one agrees to a proposal and the last two don't; The guardian who agrees can call the function `guardianCosign()` to make his vote, while the two other guardians who disagree don't have a way to make their vote rather than just doing nothing. The owner (who can call the `discardCurrentProposal(...)` function) has no way to determine if most guardians disagree. If there is no available owner, the proposal could hang forever.

`discardCurrentProposal` can be abused: The `discardCurrentPropsal()` function has the modifier `onlyOwnerOrGuardian`. This modifier allows a malicious guardian to discard any proposal without waiting for its settlement.

```
function discardCurrentProposal() external onlyOwnerOrGuardian {
    delete proposals[proposalId].newOwnerProposed;
    delete proposals[proposalId].guardiansApproved;
    delete proposals[proposalId].approvalCount;
    proposals[proposalId].resolved = true;
    emit ProposalDiscarded(proposalId);
}
```

Recommendation(s): The proposal design should **a) account for disagreeing votes** or **b) implement expiration of the proposal after a specific duration**. Additionally, Consider restricting access to the `discardCurrentProposal(...)` function to wallet owners.

Status: Fixed

Update from the client: Added in a proposal timelock mechanism. This is set to 24 hours as a default but can be changed by the wallet owner. This will allow guardians to call `discardCurrentProposal()` and as long as the current timestamp is greater than the time at which the proposal was created (stored in the `NewOwnerProposal` struct) plus the proposal timelock period, it will allow guardians to remove proposals. The call will revert if it does not comply with these parameters.

Comments: I disagree with this example: "As an example, if there are 3 guardians, one agrees to a proposal and the last two don't; The guardian who agrees can call the function `guardianCosign()` function to make his vote, while the two other guardians who disagree don't have a way to make their vote rather than just doing nothing. The owner (who can call the `discardCurrentProposal(...)` function) has no way to figure out if the majority of the guardians disagree."

If there are 3 guardians, then this would be the scenario:

1. guardian1 make the proposal;
2. guardian2 and guardian3 don't agree;
3. proposals require a 60% quorum to pass. Therefore the proposal does not pass;

The way for the owner to see if the majority of guardians agree is that the proposal passes. If the proposal does not pass (or achieve quorum), then it is self-evident that the majority of guardians don't agree.

Update from Nethermind: The timelock mechanism addresses all issues. So, we agree with the client.

6.4 [Low] In case a post user operation reverts, `_postOp(...)` function totally refunds the sponsor

File(s): src/paymaster/EtherspotPaymaster.sol

Description: In the contract `EtherspotPaymaster`, the `_postOp(...)` function is in charge of handling the user operation after it has been executed, reverted, or reverted during post-operation itself. This function contains accounting logic to refund the `sponsor` if the `prefundedAmount` is too high. When the user operation succeeds but `postOp()` call is reverted, a second call to `postOp()` is done with `PostOpMode.postOpReverted` mode. The handling of this case totally refunds the sponsor, although the user operation has succeeded and consumed gas.

```
if (mode == PostOpMode.postOpReverted) {
    ////////////////////////////////////////////////////////////////////////////////
    // @audit Should not fully refund sponsor. User operation has succeeded and consumed gas
    ////////////////////////////////////////////////////////////////////////////////
    _creditSponsor(paymaster, prefundedAmount);
    emit SponsorUnsuccessful(paymaster, sender);
} else {
```

Recommendation(s): Consider removing the current handling of `PostOpMode.postOpReverted` mode and **exclusively** refund unused gas to the sponsor (as currently done when the mode is different from `PostOpMode.postOpReverted`).

Status: Fixed

Update from the client: Removed handling of `PostOpMode.postOpReverted` and now just refunds unused gas to the sponsor.

6.5 [Low] `proposalId` 0 can be cosigned by guardians without being proposed

File(s): src/access/AccessController.sol

Description: The `proposalId` 0 can be cosigned by any guardian. This can result in approving an invalid owner address (address(0)):

```
if (
    proposals[proposalId].resolved == false
    proposals[proposalId].guardiansApproved.length != 0 &&
    ) revert("ACL:: latest proposal not yet resolved");
```

The current check in the `guardianCosign(...)` function allows signing the current proposal but does not check that the `proposalId` if above 0.

```
function guardianCosign(uint256 _proposalId) external onlyGuardian {
    require(_proposalId <= proposalId, "ACL:: invalid proposal id"); //@audit proposal id can be 0
    require(!_checkIfSigned(_proposalId), "ACL:: guardian already signed proposal");
```

Recommendation(s): Consider checking that the `_proposalId` is above 0:

```
- require(_proposalId <= proposalId, "ACL:: invalid proposal id");
+ require(_proposalId > 0 && _proposalId <= proposalId, "ACL:: invalid proposal id");
```

Status: Fixed

Update from the client: Fixed by adding the following check:

```
require(
    _proposalId != 0 && _proposalId <= proposalId,
    "ACL:: invalid proposal id"
);
```

6.6 [Info] Owners cannot remove themselves from the wallet

File(s): src/access/AccessController.sol

Description: The current implementation of `_removeOwner(...)` function disallows an owner from removing himself, even if there are other owners in the wallet. The check of `ownerCount` is sufficient to ensure that the wallet is not ownerless.

```
function _removeOwner(address _owner) internal {
    require(msg.sender != _owner, "ACL:: removing self"); //@audit an owner cannot remove himself from the wallet
    require(owners[_owner], "ACL:: non-existent owner");
    require(ownerCount > 1, "ACL:: wallet cannot be ownerless");
    emit OwnerRemoved(_owner);
    owners[_owner] = false;
    ownerCount = ownerCount - 1;
}
```

Recommendation(s): If this is not an intended behavior, consider removing the check to allow an owner to remove himself from a wallet if he is not the only owner.

Status: Fixed

Update from the client: Check for owner removing themselves has been removed.

6.7 [Info] Proposals history/traceability is not accessible from the smart contract storage

File(s): src/access/AccessController.sol

Description: When a proposal is discarded, the struct fields are deleted from the storage of the smart contract. However, this approach lacks traceability and makes it challenging to investigate historical data in case of incidents. Additionally, the `resolved` field does not give clear information if a proposal was discarded or passed through a quorum.

```
function discardCurrentProposal() external onlyOwnerOrGuardian {
    ////////////////////////////////////////////////////////////////////////////////
    // @audit Proposal fields are deleted from the storage
   ////////////////////////////////////////////////////////////////////////////////
    delete proposals[proposalId].newOwnerProposed;
    delete proposals[proposalId].guardiansApproved;
    delete proposals[proposalId].approvalCount;
    proposals[proposalId].resolved = true;
    emit ProposalDiscarded(proposalId);
}
```

```
struct NewOwnerProposal {
    address newOwnerProposed;
    uint256 approvalCount;
    address[] guardiansApproved;
    bool resolved;
}
```

Recommendation(s): To improve traceability, consider not deleting the proposal fields when it is discarded. Moreover, adding a field that denotes the type of proposal settlement (discarded or passed) is recommended.

Status: Fixed

Update from the client: When discarding proposals, the data contained in the proposal is no longer deleted.

6.8 [Info] Removed guardian's signature in the latest proposal remains valid

File(s): src/access/AccessController.sol

Description: If a guardian creates a new proposal or co-signs an existing one and is removed afterward for his malicious behavior, his proposal/ signature will still be considered in the latest proposal.

```
function removeGuardian(address _guardian) external onlyOwner {
    _removeGuardian(_guardian);
}
```

Recommendation(s): Consider discarding the current proposal when removing one of the guardians.

Status: Acknowledged

Update from the client: We are happy with this approach as it would still require a quorum of 60% of total guardians to pass, so even if there is one bad actor, it would require multiple votes to pass the proposal.

6.9 [Info] Resolved proposals can be discarded afterward

File(s): src/access/AccessController.sol

Description: The `discardCurrentProposal()` function does not check that the current proposal is `resolved` before discarding it. Once a proposal is resolved, discarding it should not be possible.

```
function discardCurrentProposal() external onlyOwnerOrGuardian {
    ////////////////////////////////////////////////////////////////////////////////
 // @audit Check that the proposal is not already resolved
 ////////////////////////////////////////////////////////////////////////////////
    delete proposals[proposalId].newOwnerProposed;
    delete proposals[proposalId].guardiansApproved;
    delete proposals[proposalId].approvalCount;
    proposals[proposalId].resolved = true;
    emit ProposalDiscarded(proposalId);
}
```

Recommendation(s): Before discarding a proposal, consider checking that the proposal is not already resolved.

Status: Fixed

Update from the client: The function `discardCurrentProposal()` now checks to see if the proposal has been resolved and the function will revert if it has.

6.10 [Info] Unnecessary inheritance in `Whitelist.sol`

File(s): src/paymaster/Whitelist.sol

Description: The `Whitelist` contract is inheriting from `Ownable` library while it does not have any owner-specific operations. The code snippet is shown below.

```
import "@openzeppelin/contracts/access/Ownable.sol";
contract Whitelist is Ownable {...}
```

Recommendation(s): Revisit the usage of `Ownable` library in the contract and consider removing the inheritance if it is not intended to be used.

Status: Fixed

Update from the client: Removed Openzeppelin `Ownable` contract import.

6.11 [Info] Unused variables in `EtherspotWallet.sol`

File(s): src/wallet/EtherspotWallet.sol

Description: Both `magicValue` and `validationData` variables are not respectively used in the functions `isValidSignature(...)` and `_validateSignature(...)`. The code snippet is reproduced below.

```
function isValidSignature(
        bytes32 hash,
        bytes calldata signature
    ) external view returns (bytes4 magicValue) {
    address owner = ECDSA.recover(hash, signature);
    if (isOwner(owner)) {
        return ERC1271_SUCCESS;
    }
    return bytes4(0xffffffff);
}
```

```
function _validateSignature(
        UserOperation calldata userOp,
        bytes32 userOpHash
    ) internal virtual override returns (uint256 validationData) {
    bytes32 hash = userOpHash.toEthSignedMessageHash();
    if (!isOwner(hash.recover(userOp.signature)))
        return SIG_VALIDATION_FAILED;
    return 0;
}
```

Recommendation(s): Remove unused variables from the returns signatures.

Status: Fixed

Update from the client: Removed `magicValue` and `validationData` return variables.

6.12 [Info] `EntryPoint` address should be immutable in `EtherspotWallet` contract

File(s): src/wallet/EtherspotWallet.sol

Description: According to the EIP-4337 standard, `EntryPoint` address contained in `EtherspotWallet` contract is expected to be hard-coded.

This component is a highly sensitive point of trust in the `EIP-4337` architecture. There should be only one entry point deployed per chain.

If a new entry point is deployed, users should update to a new version of the `EtherspotWallet` that would point to it. However, in the current implementation of the `EtherspotWallet`, the wallet owner can update the entry point address.

```
modifier onlyOwner() {
    require(
        isOwner(msg.sender) || msg.sender == address(this),
        "ACL:: only owner"
    );
    _;
}
```

```
////////////////////////////////////////////////////////////////////////////////
// @audit Owners should not be allowed to update the EntryPoint contract address
////////////////////////////////////////////////////////////////////////////////
function updateEntryPoint(address _newEntryPoint) external onlyOwner {
    require( _newEntryPoint != address(0), "EtherspotWallet:: EntryPoint address cannot be zero" );
    emit EntryPointChanged(address(_entryPoint), _newEntryPoint);
    _entryPoint = IEntryPoint(payable(_newEntryPoint));
}
```

Recommendation(s): Consider declaring `_entryPoint` variable as `immutable` and initialize its value within the constructor.

Status: Fixed

Update from the client: Passing `_entryPoint` variable value in on implementation contract construction. Removed any ability to update this variable and the only way to do so is for a new implementation contract to be deployed and a user upgrade to that new implementation.

6.13 [Info] `EtherspotWallet` can be initialized with an invalid `EntryPoint` contract address

File(s): src/wallet/EtherspotWallet.sol

Description: `EtherspotWallet` contract takes the `EntryPoint` contract address as a parameter during initialization. However, it does not perform a zero address check before assigning it.

```
function _initialize(
    address anEntryPoint,
    address anOwner
) internal virtual {
    //////////////////////////////////////////////////////////////////
    // @audit: `anEntryPoint` can be address(0)
    //////////////////////////////////////////////////////////////////
    _entryPoint = anEntryPoint;
    _addOwner(anOwner);
    emit EtherspotWalletInitialized(_entryPoint, anOwner);
}
```

Recommendation(s): Ensure that `anEntryPoint` is valid and not `address(0)`.

Status: Fixed

Update from the client: EntryPoint contract address is no longer passed in on initialization. Address checks provided for both EntryPoiny and WalletFactory (which is now passed in to provide callback to validate implementation).

6.14 [Info] `QuorumNotReached` event has an incorrect parameter name

File(s): src/access/AccessController.sol

Description: The `guardiansApproved` field in the `NewOwnerProposal` structure is an array of addresses. However, the `guardiansApproved` parameter in the `QuorumNotReached` event is declared `uint256` and represents the approval count. Using the same name within the structure and the event can be confusing. The code snippet is reproduced below.

```
struct NewOwnerProposal {
    address newOwnerProposed;
    uint256 approvalCount;
    address[] guardiansApproved;
    bool resolved;
};
```

```
event QuorumNotReached(
    uint256 proposalId,
    address newOwnerProposed,
    uint256 guardiansApproved
);
```

Recommendation(s): Consider changing the parameter name.

Status: Fixed

Update from the client: Renamed `guardiansApproved` in `QuorumNotReached` event to `approvalCount`.

6.15 [Best Practice] `onlyOwner` modifier name represents incorrect logic

File(s): src/access/AccessController.sol

Description: The contract has three modifiers: `onlyOwnerOrEntryPoint`, `onlyOwnerOrGuardian`, and `onlyOwner`. The `onlyOwner` and `onlyOwnerOrGuardian` modifiers return `true` when the caller is `address(this)`. However, `onlyOwnerOrEntryPoint` returns `false` in that case which can lead to confusion.

Recommendation(s): Consider changing the modifier name to make it more explicit.

Status: Acknowledged

Update from the client: We have decided to keep this as is, as this reflects the current implementation in Infintism's codebase.

6.16 [Best Practices] Apply Checks-Effects-Interactions pattern

File(s): src/paymaster/EtherspotPaymaster.sol

Description: The function `depositFunds(...)` violates the Checks Effects Interactions pattern as it makes an external call to `entryPoint`, followed by a state update of the sponsor funds (in `_creditSponsor()` function). The code is reproduced below.

```
function depositFunds() external payable nonReentrant {
    entryPoint.depositTo{value: msg.value}(address(this));
    _creditSponsor(msg.sender, msg.value);
}
```

Recommendation(s): Apply the Checks-Effects-Interactions pattern by placing the state update before the external call.

```
function depositFunds() external payable nonReentrant {
+   _creditSponsor(msg.sender, msg.value);
    entryPoint.depositTo{value: msg.value}(address(this));
-   _creditSponsor(msg.sender, msg.value);
}
```

Status: Fixed

Update from the client: Applied Checks-Effects-Interactions pattern.

6.17 [Best Practices] Contract `Whitelist` does not use `IWhitelist.sol`

File(s): src/paymaster/Whitelist.sol

Description: The contract `Whitelist.sol` does not import the interface `IWhitelist.sol`, resulting in not using its functionalities. Additionally, some functions and events are declared in the interface but are not used in the contract.

```
////////////////////////////////////////////////////////////
// @audit Whitelist.sol is not using its interface
////////////////////////////////////////////////////////////
import "@openzeppelin/contracts/access/Ownable.sol";

contract Whitelist is Ownable {...}
```

Recommendation(s): To avoid duplicated code, consider importing `IWhitelist.sol` interface in `Whitelist.sol` contract and remove unused events or functions from the interface declaration.

Status: Fixed

Update from the client: Imported `IWhitelist.sol` interface in `Whitelist.sol`. Removed unused events.

6.18 [Best Practices] Functions that can have `external` visibility

File(s): src/wallet/EtherspotWallet.sol, src/wallet/EtherspotWalletFactory.sol

Description: The following functions have `public` visibility instead of `external`. External functions and public functions are both used to allow external access to contract functions. However, there are some key differences between them. External functions are more gas-efficient compared to public functions. Moreover, external functions can only be called from outside the contract. This restriction can enhance security by preventing potential reentrancy attacks and unintended recursive calls that can be exploited to manipulate the contract state. Finally, using external functions improves the readability of the code. The functions that can be made `external` are listed below.

```
- EtherspotWalletFactory.createAccount(...)
- EtherspotWallet.withdrawDepositTo(...)
```

Recommendation(s): Set these functions' visibility as `external`.

Status: Fixed

Update from the client: Set visibility of `EtherspotWalletFactory.createAccount(...)` and `EtherspotWallet.withdrawDepositTo(...)` to external.

6.19 [Best Practices] Interface and implementation are not matching in `EtherspotWallet.sol`

File(s): `src/wallet/EtherspotWallet.sol`

Description: The contract `EtherspotWallet.sol` does not import the interface `IEtherspotWallet.sol`. The interface defines 5 events, while the contract only defines 3 events. The code snippets are shown below.

```
////////////////////////////////////////////////////////////
// @audit Interface "IEtherspotWallet" defines 5 events
////////////////////////////////////////////////////////////
event EtherspotWalletInitialized(
    IEntryPoint indexed entryPoint,
    address indexed owner
);
event EtherspotWalletReceived(address indexed from, uint256 indexed amount);
event EntryPointChanged(address oldEntryPoint, address newEntryPoint);
event OwnerAdded(address newOwner, uint256 blockFrom);
event OwnerRemoved(address removedOwner, uint256 blockFrom);
```

```
////////////////////////////////////////////////////////////
// @audit Contract "EtherspotWallet" defines only three events
////////////////////////////////////////////////////////////

event EtherspotWalletInitialized(
    IEntryPoint indexed entryPoint,
    address indexed owner
);
event EtherspotWalletReceived(address indexed from, uint256 indexed amount);
event EntryPointChanged(address oldEntryPoint, address newEntryPoint);

////////////////////////////////////////////////////////////
// @audit Contract is not IEtherspotWallet
////////////////////////////////////////////////////////////
contract EtherspotWallet is
    BaseAccount,
    UUPSUpgradeable,
    Initializable,
    TokenCallbackHandler,
    AccessController,
    IERC1271Wallet
{...}
```

Recommendation(s): Remove contradictions between the interface and the contract.

Status: Fixed

Update from the client: Removed unused/unnecessary events.

6.20 [Best Practices] Remove unused `UniversalSignatureValidator` contract from the codebase

File(s): `src/helpers/UniversalSignatureValidator.sol`

Description: The `UniversalSignatureValidator` contract is not used anywhere in the codebase of the Etherspot Wallet project. It seems to be a remainder from a previous version.

Recommendation(s): Remove unused contracts from the codebase.

Status: Acknowledged

Update from the client: We have decided to keep this contract in the repo as it will be used at some point in the future.

6.21 [Best Practices] Unclear function names in the `Whitelist` Contract

File(s): src/paymaster/Whitelist.sol

Description: The current function names in the `Whitelist` contract are unclear in conveying that they relate to whitelist functionalities. The functions `add(...)`, `addBatch(...)`, `remove(...)`, and `removeBatch(...)` perform operations on the whitelist, but their names do not explicitly indicate this. These functions' names should be more specific, as the `Whitelist` contract will be inherited and exposed by the `EtherspotPaymaster` contract.

Recommendation(s): To improve clarity, it is suggested to rename the functions to reflect better their purpose related to the whitelist, e.g., `addToWhitelist(...)`, `addBatchToWhitelist(...)`, `removeFromWhitelist(...)`, `removeBatchFromWhitelist(...)`.

Status: Fixed

Update from the client: Amended function names to `addToWhitelist(...)`, `addBatchToWhitelist(...)`, `removeFromWhitelist(...)`, `removeBatchFromWhitelist(...)`.

6.22 [Best Practices] Unnecessary usage of input parameter

File(s): src/paymaster/EtherspotPaymaster.sol

Description: The function `withdrawFunds(...)` takes `address payable _sponsor` as an input. However, it checks the funds and performs the withdrawal of `msg.sender` as he/she is the only one allowed to withdraw his/hers funds. The input is only checked to equal the `msg.sender` variable. The withdrawal can be exclusively based on the `msg.sender` variable instead of using the `_sponsor` parameter. The code snippet is shown below.

```
function withdrawFunds(
    address payable _sponsor,
    uint256 _amount
) external nonReentrant {
////////////////////////////////////////////////////////////
// @audit _sponsor parameter is unnecessary
////////////////////////////////////////////////////////////
    require(
        msg.sender == _sponsor,
        "EtherspotPaymaster:: can only withdraw own funds"
    );
    require(
        checkSponsorFunds(_sponsor) >= _amount,
        "EtherspotPaymaster:: not enough deposited funds"
    );
    _debitSponsor(_sponsor, _amount);
    entryPoint.withdrawTo(_sponsor, _amount);
}
```

Recommendation(s): Consider removing the `_sponsor` parameter and using the `msg.sender` transaction variable.

Status: Fixed

Update from the client: Removed `_sponsor` function parameter in favour of using `msg.sender`.

6.23 [Best Practices] Unused `WhitelistInitialized` event in `Whitelist.sol`

File(s): src/paymaster/Whitelist.sol

Description: `WhitelistInitialized` event in `Whitelist.sol` is never emitted.

```
////////////////////////////////////////////////////////////
// @audit Event not used
////////////////////////////////////////////////////////////
event WhitelistInitialized(address owner);
```

Recommendation(s): Check if you need the event. In case you don't need it, remove any unused event declarations.

Status: Fixed

Update from the client: Removed.

6.24 [Best Practices] `NewOwnerProposal` struct is not reordered to consume less gas

File(s): src/access/AccessController.sol

Description: The struct `NewOwnerProposal` can be reordered to fit fewer slots. This reordering saves gas when storing new structure data in the smart contract storage.

```
////////////////////////////////////////////////////////////////////////////////////////////////
// @audit: Struct order can be changed to (address, bool, uint256, address[]) to fit in 1 slot less.
////////////////////////////////////////////////////////////////////////////////////////////////
struct NewOwnerProposal {
    address newOwnerProposed;
    uint256 approvalCount;
    address[] guardiansApproved;
    bool resolved;
}
```

Recommendation(s): Reorder struct properties like following.

```
struct NewOwnerProposal {
 address newOwnerProposed;
+ bool resolved;
 uint256 approvalCount;
 address[] guardiansApproved;
- bool resolved;
}
```

Status: Fixed

Update from the client: Rearranged variables instruction to:

```
struct NewOwnerProposal {
    address newOwnerProposed;
    bool resolved;
    uint256 approvalCount;
    address[] guardiansApproved;
}
```

6.25 [Best Practices] `accountImplementation` cannot be updated

File(s): src/wallet/EtherspotWalletFactory.sol

Description: The `accountImplementation` variable stores the implementation address used to deploy new wallets. This variable is declared `immutable` in `EtherspotWalletFactory.sol`. It cannot be updated to set newer implementations.

```
contract EtherspotWalletFactory {
////////////////////////////////////////////////////////
// @audit: accountImplementation cannot be updated
////////////////////////////////////////////////////////
address public immutable accountImplementation;

constructor() {
    accountImplementation = address(new EtherspotWallet());
}
```

Recommendation(s): Consider removing the immutable attribute to the `accountImplementation` variable and adding accessors functions to update it.

Status: Fixed

Update from the client: Removed `immutable` keyword and added function for updating `accountImplementation`.

6.26 [Best Practices] checkSponsorFunds name is confusing

File(s): src/paymaster/EtherspotPaymaster.sol

Description: The function checkSponsorFunds(...) within the EtherspotPaymaster.sol contract is a getter for the mapping sponsorFunds and does not check any specific condition. The current name can lead to misunderstandings.

Recommendation(s): Consider renaming the function to a more descriptive name that accurately reflects its role.

Status: Fixed

Update from the client: Renamed mapping from sponsorFunds to _sponsorBalances. Renamed checkSponsorFunds() to getSponsorBalance().

7 Etherspot Wallet upgrade architecture

7.1 Current `EtherspotWallet` upgrade architecture

The `EtherspotWalletFactory` is in charge of creating a new instance of the `Proxy`. The proxy follows the EIP-1967 standard and its implementation slot points to the `EtherspotWallet` implementation contract. This design has the following advantages and disadvantages.

Pros:

- This architecture is gas efficient as each new account only needs to deploy the minimal `Proxy` contract;
- Upgrade of the implementation follows the Universal Upgradeable Proxy Standard (`UUPS`) EIP-1822.

Cons:

- This architecture does not restrict the value of the updated implementation of the `EtherspotWallet`;
- With the current pattern, the new wallets are always obliged to deploy with the first implementation (the immutable one), then upgrade to a new one;
- The current `EtherSpotWalletFactory` contract cannot update the `accountImplementation` variable (currently set as immutable).

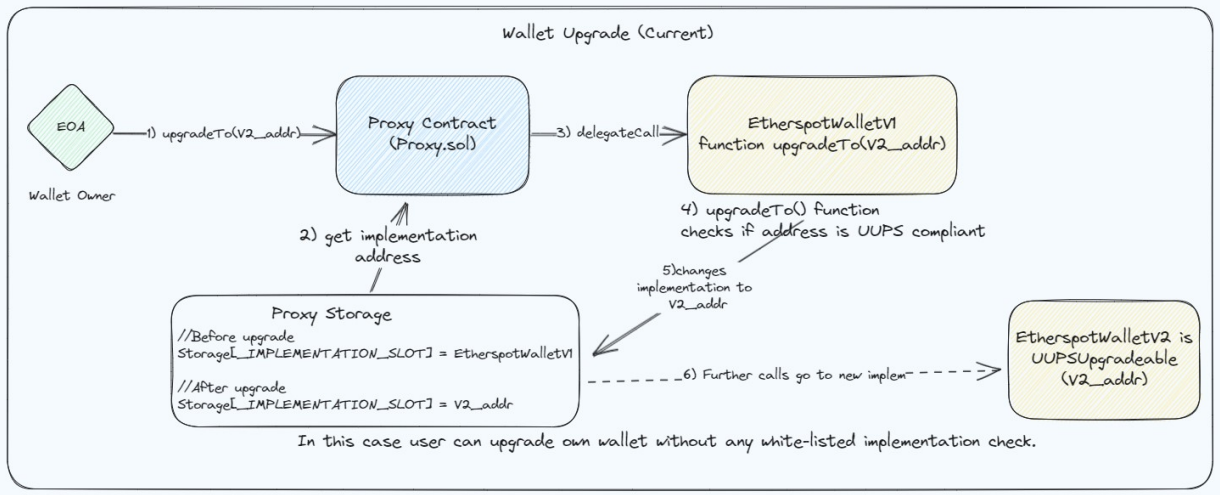


Fig. 2: Current Wallet Upgrade

The upgrade of `EtherspotWallet` is illustrated in Fig. 2. The process of upgrading the `EtherspotWallet` is currently done as follows:

a. User triggers an update of the implementation wallet calling `upgradeTo(...)` providing the address of the new `EtherspotWallet` implementation (`V2_addr` in the diagram).

b. `Proxy` contract retrieves the current `EtherspotWallet` implementation address into its storage slot.

c. `Proxy` contract forwards (`delegatecall()`) this call to the current `EtherspotWallet` implementation contract.

d. `EtherspotWallet` implementation follows the `UUPS` standard. The `upgradeTo(...)` function checks the new implementation (`V2_addr`) to be `UUPS` compliant.

e. If provided implementation address follows the `UUPS` standard, `Proxy` implementation slot is updated to the provided address (`V2_addr`).

f. Further calls to the `Proxy` are delegated to the new implementation address.

7.2 Suggested upgrade architecture

The current upgrading scheme of `EtherspotWallet` is well designed but needs a few improvements to limit updates to a **restricted set** of `EtherspotWallet` implementations. One constraint is that the user can only trigger the upgrade of the `EtherspotWallet` implementation. To restrict the upgrade of the `EtherspotWallet` to a set of contracts, the upgrade mechanism must query an external-and-trusted smart contract (`EtherspotWalletFactory`, in our case) before allowing the upgrade. The suggested process is illustrated in Fig. 3. The green text and green arrows represent the additional steps/actions that should be done to restrict the upgrade to a set of whitelisted `EtherspotWallet` implementation addresses. The suggested upgrade process of the `EtherspotWallet` would be as follows:

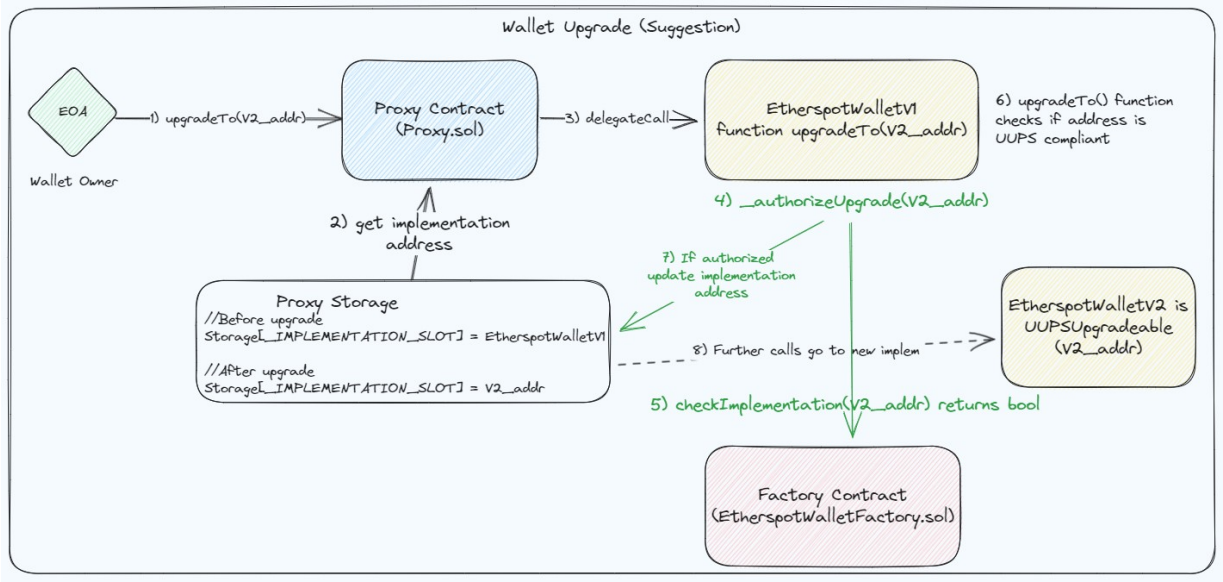


Fig. 2: Suggested Wallet Upgrade

1. User triggers an update of the implementation wallet calling `upgradeTo(...)` providing the address of the new `EtherspotWallet` implementation (`V2_addr` in the diagram).
2. `Proxy` contract retrieves the current `EtherspotWallet` implementation address from its storage slot.
3. `Proxy` contract forwards (`delegatecall()`) this call to the current `EtherspotWallet` implementation contract.
4. The `UUPS` standard provides the `_authorizeUpgrade(...)` function that can be overridden by the implementing contract (`EtherspotWallet` in our case) to add new checks before triggering the update. This can be used to **restrict** the provided implementation address.
5. The `_authorizeUpgrade(...)` would make an external call (`checkImplementation(V2_addr)`) to the `EhterspotWalletFactory` contract with the address provided by the Wallet's owner. `EtherspotWalletFactory` contract would return a boolean, indicating that the provided contract address is whitelisted or not (returning respectively `true` or `false`).
6. If contract is authorized, The `upgradeTo(...)` function checks that the new implementation (`V2_addr`) is `UUPS` compliant.
7. Then, if provided implementation address follows the `UUPS` scheme, it is stored in the `Proxy` implementation slot.
8. Further calls to the `Proxy` are delegated to the new implementation address.

8 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future. **The documentation of the EtherSpot Wallet is presented in inline comments over the code, and also in the folder docs**. The folder presents documentation for the `Access Controller`, `EtherSpot Paymaster`, `EtherSpot Wallet`, and `WhiteList`.

9 Test Suite Evaluation

9.1 Contracts Compilation

```
> npx hardhat compile
```

```
Generating typings for: 54 artifacts in dir: typings for target: ethers-v5
Successfully generated 152 typings!
Compiled 41 Solidity files successfully
```

9.2 Tests Output

```
> npx hardhat test
```

```
  Factory
     should get counter factual address (86ms)
     counter-factual address should match deployed address (343ms)
     should return address if wallet is already deployed

  EntryPoint with EtherspotPaymaster
    #parsePaymasterAndData
       should parse data properly
    whitelist integration check
       should be able to interact with whitelist (137ms)
    #validatePaymasterUserOp
       should reject on no signature (88ms)
       should reject on invalid signature (55ms)
       succeed with valid signature (245ms)
       should reject if not a whitelisted signature (63ms)
       succeeds if whitelisted signature (206ms)
       error thrown if sponsor balance too low (195ms)
      with wrong signature
         should return signature error (no revert) on wrong signer signature (38ms)
         handleOp revert on signature failure in handleOps
    #depositFunds
       should succeed in depositing funds (151ms)
    #withdrawFunds
       should succeed in withdrawing funds (300ms)
       should throw error when sponsor is not withdrawer of funds
       should throw error when amount is greater than sponsor deposited balance
    #_postOp
       should credit remaining prefund after gas on successful UserOp execution (46ms)
       should credit whole prefund back on unsuccessful UserOp execution (39ms)
       should emit success event upon deducting sponsor funds
  EtherspotWallet
     should deploy wallet (302ms)
     owner should be able to call execute (527ms)
     owner should be able to call executeBatch (140ms)
     a different owner should be able to call execute (204ms)
     guardian should not be able to call execute (383ms)
     other account should not be able to call execute (274ms)
     should pack in js the same as solidity
    #validateUserOp
       should pay
       should return NO_SIG_VALIDATION on wrong signature
    #validateUserOp - multiple account owners
       should pay
       should return NO_SIG_VALIDATION on wrong signature
    #updateEntryPoint
       should update EntryPoint contract address (105ms)
       should trigger error if zero address passed in
    #isOwner
       should return true for valid owner
       should return false for invalid owner
```

```
  #addOwner
    should add a new owner (from owner) (108ms)
    should increment owner count (289ms)
    should emit event on new owner added (149ms)
    should trigger error if caller is not owner
    should trigger error if already owner
  #removeOwner
    should remove an owner (from owner) (314ms)
    should remove an owner (from guardian) (386ms)
    should decrement owner count (387ms)
    should emit event on removal of owner (202ms)
    should trigger error caller is not owner (135ms)
    should trigger error if removing self (180ms)
    should trigger error if removing a non owner
    should trigger error if removing owner would make wallet ownerless (171ms)
  #isGuardian
    should return true for valid guardian (195ms)
    should return false for invalid guardian
  #addGuardian
    should add a new guardian (186ms)
    should increment guardian count (291ms)
    should trigger error on zero address
    should trigger error on adding existing guardian (128ms)
    should emit event on adding guardian (133ms)
  #removeGuardian
    should remove guardian (400ms)
    should decrement guardian count (311ms)
    should trigger error on removing non-existant guardian
    should emit event on removing guardian (243ms)
Proposing, cosigning and deleting
  #getProposal
    should return proposal data for a specified proposal id (628ms)
    should trigger error if specified proposal id is invalid
  #guardianPropose
    should allow guardian to propose a new owner (246ms)
    should emit event on submitting proposal (160ms)
    should only allow guardian to call (owner can just add new owner)
    requires minimum of 3 guardians to propose a new owner (251ms)
    should only allow one active proposal at a time and throw error if trying to add another (655ms)
    should allow a new proposal after the previous one has been resolved (cosigned) (584ms)
    should allow a new proposal after the previous one has been resolved (discarded) (453ms)
  #guardianCosign
    should allow guardian to cosign proposal and not reach quorum (emits event) (415ms)
    should allow guardian to cosign proposal and reach quorum 2/3 (adds owner) (454ms)
    should allow guardian to cosign proposal and reach quorum 3/4 (adds owner) (603ms)
    should allow guardian to cosign proposal and reach quorum 3/5 (adds owner) (748ms)
    should allow guardian to cosign proposal and reach quorum 4/6 (adds owner) (1075ms)
    should only allow guardian to call (owner can just add new owner) (166ms)
    shouldn't allow guardians to approve proposal more than once (234ms)
    should throw error is invalid proposal id
  #discardCurrentProposal
    should discard current proposal (320ms)
    should emit event on proposal discard (419ms)
    should only allow owner or guardian to call (196ms)
    should not allow cosigning a discarded proposal (297ms)
EtherspotWalletFactory
  sanity: check deployer (284ms)


81 passing (34s)
```

9.3 Code Coverage

```
> npx hardhat coverage
```

The relevant output is presented below.

```
----------------------------------|----------|----------|----------|----------|----------------|
File                              |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------------------|----------|----------|----------|----------|----------------|
 access/                          |    95.74 |    81.82 |      100 |    98.51 |                |
  AccessController.sol            |    95.74 |    81.82 |      100 |    98.51 |            206 |
 helpers/                         |        0 |        0 |        0 |        0 |                |
  UniversalSignatureValidator.sol |        0 |        0 |        0 |        0 |... 121,122,127 |
 interfaces/                      |      100 |      100 |      100 |      100 |                |
  IERC721Wallet.sol               |      100 |      100 |      100 |      100 |                |
  IEtherspotPaymaster.sol         |      100 |      100 |      100 |      100 |                |
  IEtherspotWallet.sol            |      100 |      100 |      100 |      100 |                |
  IWhitelist.sol                  |      100 |      100 |      100 |      100 |                |
 paymaster/                       |    62.96 |    55.88 |    66.67 |    66.67 |                |
  BasePaymaster.sol               |       40 |       25 |    44.44 |    41.67 |... 8,84,92,101 |
  EtherspotPaymaster.sol          |    76.92 |    72.22 |    90.91 |    80.65 |... 198,200,204 |
  Whitelist.sol                   |    55.56 |       50 |       60 |       60 |... 72,73,87,88 |
 wallet/                          |    80.49 |    53.33 |    71.43 |    81.48 |                |
  EtherspotWallet.sol             |    70.83 |       50 |    64.29 |    70.97 |... 145,162,181 |
  EtherspotWalletFactory.sol      |    93.33 |    66.67 |       80 |    94.74 |             25 |
  Proxy.sol                       |      100 |       50 |      100 |      100 |                |
----------------------------------|----------|----------|----------|----------|----------------|
All files                         |     65.5 |    60.14 |    73.33 |    69.72 |                |
----------------------------------|----------|----------|----------|----------|----------------|
```

9.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.