Security Review Report
NM-0697 Hyperbeat Management Account

NETHERMIND SECURITY

(December 17, 2025)

Contents

1 Executive Summary

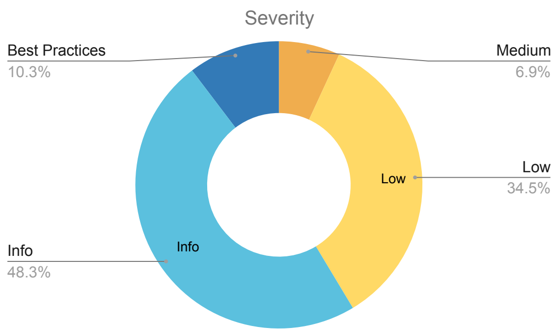
This document presents the results of a security review conducted by Nethermind Security for Hyperbeat Management Account contracts. The protocol implements a sophisticated, non-custodial smart contract wallet ecosystem centered around **ManagementAccount**. Each user is assigned a unique ManagementAccount, which is deployed and overseen by a central **ManagementAccountFactory** contract. The factory contract serves as the system's backbone with global configurations such as the underlying wallet implementation logic, system-wide security parameters like delays and fees, and registries for approved DeFi integrations and whitelisted tokens. This centralized deployment ensures that all user accounts adhere to a standardized, secure framework while maintaining individual ownership.

It allows users to maintain full custody of their crypto assets while enabling card payment settlements through an authorized operator.

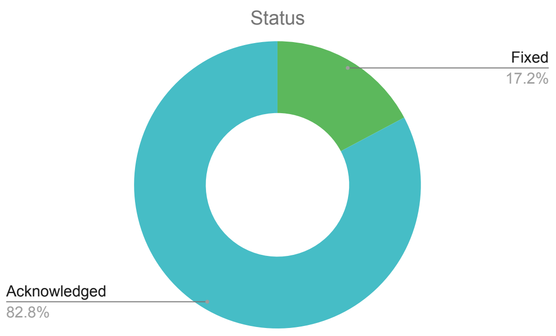
The wallet operates in spending or credit mode at the choice of the wallet owner. While deposits in the wallet are free, withdrawals can attract fees, which is configured by the protocol owner. The wallet implements criteria for withdrawal of `settlement token` based on the currently active mode.

The audit comprises 1785 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools. **Along this document, we report** 29 points of attention. where two are classified as `Medium`, ten are classified as `low` and seventeen are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.



(a)



(b)

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (10), **Undetermined** (0), **Informational** (14), **Best Practices** (3).
Distribution of status: Fixed (5), **Acknowledged** (24), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	November 25, 2025
Final Report	December 17, 2025
Initial Commit	09e970
Final Commit	33f200c
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	core/ManagementAccountStorage.sol	36	4	11.1%	9	49
2	core/ManagementAccountFactory.sol	147	17	11.6%	27	191
3	core/ManagementAccount.sol	741	56	7.6%	169	966
4	services/morpho/MorphoService.sol	232	62	26.7%	42	336
5	services/registry/ServiceRegistry.sol	55	1	1.8%	20	76
6	services/registry/TokenWhitelistRegistry.sol	57	17	29.8%	22	96
7	interfaces/core/IManagementAccount.sol	123	146	118.7%	74	343
8	interfaces/core/IManagementAccountFactory.sol	12	11	91.7%	9	32
9	interfaces/services/IServiceRegistry.sol	30	1	3.3%	13	44
10	interfaces/services/ITokenWhitelistRegistry.sol	10	16	160.0%	7	33
11	interfaces/services/IService.sol	51	52	102.0%	14	117
12	libraries/ManagementAccountErrors.sol	64	7	10.9%	7	78
13	libraries/ManagementAccountLib.sol	205	19	9.3%	32	256
14	libraries/ServiceErrors.sol	6	1	16.7%	1	8
15	libraries/services/ServiceRegistryErrors.sol	8	1	12.5%	1	10
16	libraries/services/TokenWhitelistRegistryErrors.sol	8	14	175.0%	5	27
	Total	**1785**	**425**	**23.8%**	**452**	**2662**

3 Summary of Issues

	Finding	Severity	Update
1	Flawed market routing in MorphoService allows for market hijacking	Medium	Acknowledged
2	Possible to authorize borrowing for multiple services	Medium	Fixed
3	Retroactive Fee Application in Withdrawal Processing	Low	Fixed
4	Approvals granted in credit mode are not revoked when switching to spending mode	Low	Acknowledged
5	Extended Wait Time for Settlement Token Withdrawals Due to two-step Approval Process	Low	Acknowledged
6	Fee calculations round in favor of users instead of the protocol	Low	Acknowledged
7	ManagementAccount contract does not support native tokens	Low	Acknowledged
8	Old Operator Authorization Not Revoked When Operator Changes	Low	Acknowledged
9	Settlement token in ManagementAccountFactory can be inconsistent with TokenWhitelistRegistry	Low	Fixed
10	The `ManagementAccount` contract uses `transfer` and `transferFrom` which may fail for non-standard ERC20 tokens	Low	Fixed
11	The `MorphoService` approves tokens without zeroing allowance first causing reverts for USDT-like tokens	Low	Fixed
12	The `executeModeChange()` function fails to revoke approvals when switching credit services	Low	Acknowledged
13	`registerMarket` in MorphoService allows registering non-existent Morpho Blue markets	Info	Acknowledged
14	Approved service actions can execute without re-checking registry or owner approval status	Info	Acknowledged
15	Approved service actions cannot be rescinded and never expire	Info	Acknowledged
16	Misleading error name when processing an already cancelled withdrawal	Info	Acknowledged
17	The `_pendingMode` state variable is not cleared after execution or cancellation	Info	Acknowledged
18	The `cancelModeChange(...)` function clears pending credit service even when it's empty	Info	Acknowledged
19	The `executeServiceAction()` function bypasses token whitelist validation during the operator approval flow	Info	Acknowledged
20	The `executeServiceCalls()` function in `ManagementAccountLib` contains redundant parameters	Info	Acknowledged
21	The `getAvailableBalance()` function duplicates the logic of `_availableBalance()`	Info	Acknowledged
22	The `getBalance()` function silently suppresses errors from external services	Info	Acknowledged
23	The `requestSpendingMode()` and `requestCreditMode()` functions allow overwriting pending requests	Info	Acknowledged
24	The codebase contains unused custom error definitions	Info	Acknowledged
25	The isServiceActive(...) and isServiceValid(...) functions in ServiceRegistry are identical	Info	Acknowledged
26	`getBalance` function in MorphoService returns an incorrect value	Info	Acknowledged
27	Inconsistent naming convention in contracts	Best Practices	Acknowledged
28	The `ManagementAccountFactory` contract does not inherit from `IManagementAccountFactory`	Best Practices	Acknowledged
29	The `ManagementAccountFactory` contract inherits from non-upgradeable AccessControl	Best Practices	Acknowledged

4 System Overview

The `ManagementAccount` is a non-custodial smart contract wallet deployed by `ManagementAccountFactory` contract. The owner of the protocol manages the services available to the wallet by whitelisting them in `ServiceRegistry` contract. Similarly, the tokens supported by the wallet are also whitelisted in the `TokenWhitelistRegistry` contract by the protocol owner. The wallet only supports ERC20 tokens. Native token is not supported by the wallet.

The protocol owner nominates and manages a `settlementToken` which is applicable to all the wallets.

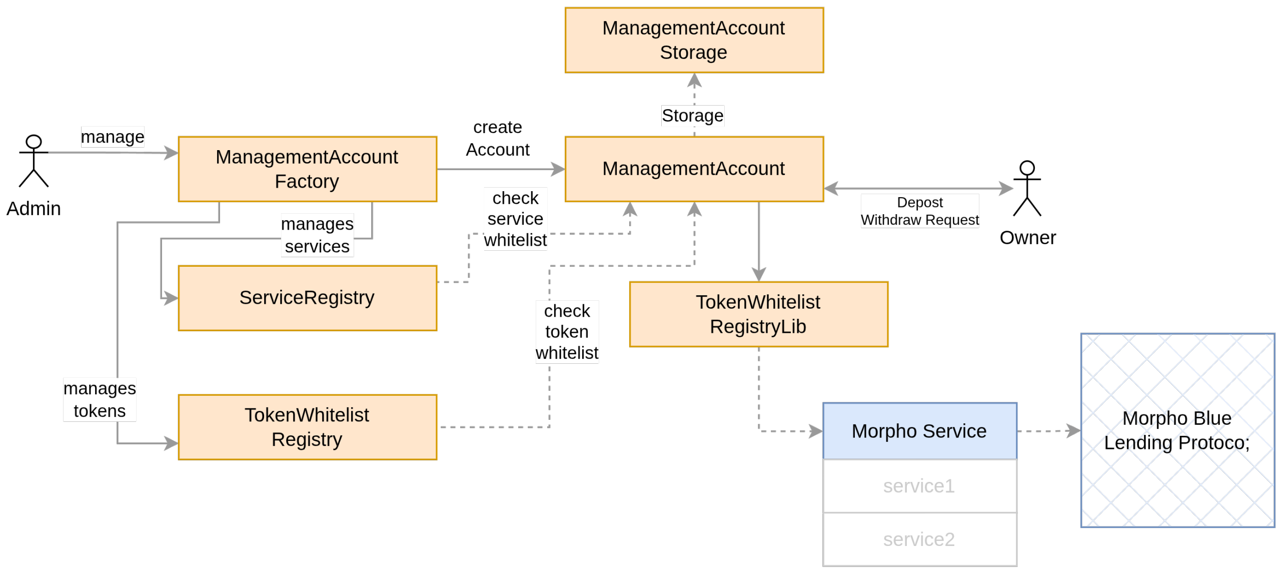


Fig. 2: Hyperbeat Management Account overview

The `ManagementAccount` wallet operates in two distinct modes.

- **SPENDING**: In `SPENDING` mode, the account acts as a secure repository to hold and transfer assets. It features a settlement allowance mechanism, enabling a designated Operator to debit a specific amount of a settlement token to facilitate off-chain transactions. For added security, withdrawals of this settlement token utilize a time-delayed, two-step approval process involving both the user and the Operator.
- **CREDIT**: The wallet in `CREDIT` mode transforms the account into a DeFi interface. Through a standardized interface, the wallet can connect to approved lending protocols (such as Morpho Blue) to deposit assets as collateral. In this state, the Operator is authorized to borrow against the user's collateral, while critical actions such as withdrawing that collateral require Operator approval to protect the solvency of the position.

4.1 Actors and Main entry points

There are two primary actors in the `ManagementAccount` workflows.

4.1.1 Owner

An account that controls the account, approves services, and manages assets held in the `ManagementAccount` wallet.

a) **deposit**: Deposits whitelisted tokens into the account from the owner's address

b) **requestWithdrawal**: Requests a withdrawal, executes instantly if no cooldown, otherwise creates a pending withdrawal request.

c) **approveService**: Approves a registered service for use by the account.

d) **executeServiceAction**: Executes service actions (deposit, withdraw, borrow, repay, custom) on approved services, may require operator approval for certain actions.

e) **changeMode**: Has functions to change the mode of the ManagementAccount between spending and credit.

f) **increaseSettlementAllowance**: Increases the settlement token spending allowance for the operator.

g) **authorizeOperatorBorrowing**: Authorizes the operator to borrow on behalf of the account in CREDIT mode.

4.1.2 Operator

Operator is an account configured by the owner of the protocol with authorization to perform actions on behalf of the owner of `ManagementAccount` wallet, if approved to do so.

a) **settle**: Withdraws settlement tokens from the account, consuming from the spending limit.

b) **approveWithdrawal**: Approves a pending withdrawal request for settlement token in SPENDING mode after cooldown.

c) **processWithdrawal**: Processes a single withdrawal request that has passed cooldown and approval requirements.

d) **executeModeChange**: Executes a pending mode change after the cooldown period, enforces settlement balance requirements when switching to SPENDING mode.

e) **approveServiceAction**: Approves a pending service action that requires operator approval.

f) **approveDecreaseSettlementAllowance**: Approves a pending settlement allowance decrease request.

The protocol implements a dual-control security model by delineating specific roles for the account Owner and the system Operator. This separation prevents unilateral action on critical functions, requiring distinct authorization for sensitive tasks such as withdrawals, allowance modifications, and mode switching.

4.2 Interactions Flows

The protocol exposes several key functions for state modification that involves both owner and operator. They typically follow a request-approve-execute pattern:

a) **Withdrawal**: Owner initiates the withdrawal process. If there is no cool down period, it executes immediately. If cool down period applies, operator has to approve before the withdrawal can be complete.

b) **Mode Change**: Owner initiates the mode change request which will wait for a delay period. After expiry of the deplay period, the operator executes the mode change.

a) **Settlement Allowance**: Owner can increase the settlement allowance any time. But decreasing allowance needs approval from Operator. Post approval, owner can execute change in settlement allowance.

a) **Service Action**: Service action workflow depends on the type of action. Some actions does not needs approval from operator. Those actions are executed immediately by the owner, while others goes through the operator approval process.

5 Risk Rating Methodology

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

Likelihood measures 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 an attacker exploits the finding. 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 Security 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 pass to the client formally;

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] Flawed market routing in MorphoService allows for market hijacking

File(s): MorphoService.sol

Description: The `MorphoService` contract facilitates interactions with Morpho Blue markets. To route user actions—such as depositing collateral or borrowing assets—to the correct market, the contract utilizes two mappings: `collateralMarket` and `loanMarket`. These mappings are designed to associate a specific token address with its corresponding market ID.

The `registerMarket()` function is used by the protocol's `owner` to onboard new markets. When executed, it updates the mappings to associate the market's collateral and loan tokens with the new market ID.

However, this design assumes a one-to-one relationship between a `token` and a `market`, which is incorrect in the context of Morpho Blue. A single asset (e.g., WETH) can function as collateral in multiple distinct markets (e.g., WETH/USDC and WETH/DAI).

If the owner registers a second market that uses a token already registered for a previous market, the `registerMarket()` function will overwrite the existing mapping. For instance, registering a WETH/DAI market after a `WETH/USDC` market will change collateralMarket[WETH] to point to the `WETH/DAI` market ID.

Consequently, any subsequent user interaction involving `WETH` as collateral will be blindly routed to the `WETH/DAI` market, even if the user intended to interact with their existing `WETH/USDC` position. This flawed routing can cause users to deposit funds into the wrong market or prevent them from managing existing positions, leading to potential fund lockups.

```
function registerMarket(MarketParams memory params) external onlyOwner {
    Id marketId = params.id();
    require(!isMarketRegistered[marketId], "MorphoService: already registered");
    // ...
    marketParamsById[marketId] = params;
    isMarketRegistered[marketId] = true;

    // @audit-issue This overwrites the mapping if the token is used in another market.
    collateralMarket[params.collateralToken] = marketId;
    // @audit-issue This overwrites the mapping if the token is used in another market.
    loanMarket[params.loanToken] = marketId;

    emit MarketRegistered(marketId, params.collateralToken, params);
}
```

Recommendation(s): Consider refactoring the routing logic to support a one-to-many relationship between tokens and markets. Instead of relying on global mappings to infer the market ID from a token address, consider requiring callers to pass the specific marketId as a parameter for functions dealing with deposits or borrows.

Status: Acknowledged

Update from the client: It's like entended the service will be used with one market for specific collateral token for credit mode since loan token is always the same (settlement token). And for regular usage we can deploy multiple morpho services and approve them

6.2 [Medium] Possible to authorize borrowing for multiple services

File(s): ManagementAccount.sol

Description: The `ManagementAccount` owner can authorize an operator to borrow funds on their behalf when the account is in `CREDIT` mode. This is done by calling the `authorizeOperatorBorrowing()` function, which specifies a `service`, an `authType`, and an `amount`.

When this authorization is set, the function stores the `authType` in the singular state variable `_creditAuthorizationType`.

```
function authorizeOperatorBorrowing( address service, IService.AuthorizationType authType,
uint256 amount
)
    external
    override
    onlyOwner
    nonReentrant
    returns (bytes[] memory results)
{
    if (_mode != AccountMode.CREDIT) {
        revert ManagementAccountErrors.OperatorBorrowingRequiresCreditMode();
    }
    // ...
    // @audit-issue This is a single state variable.
    _creditAuthorizationType = authType;
}
```

The issue is that `_creditAuthorizationType` is not mapped per service. If the owner calls `authorizeOperatorBorrowing()` for a second service, this state variable will be overwritten with the `authType` of the new service.

This variable is later read by the `approveAuthorizationRevocation()` function to build the revocation calls.

```
function approveAuthorizationRevocation() external onlyOperator nonReentrant {
    if (_pendingAuthRevocationService == address(0)) {
        revert ManagementAccountErrors.NoAuthRevocationPending();
    }
    address service = _pendingAuthRevocationService;
    // ...
    // @audit-issue Reads the last saved `_creditAuthorizationType`.
    IService.Call[] memory calls = IService(service).buildRevokeAuthorization(
        _creditAuthorizationType,
        operatorAddress,
        settlementTokenAddress
    );

    // ...
}
```

This will lead to incorrect behaviour when revoking authorizations. If the owner authorizes borrowing from Service A (with authType 1) and then Service B (with authType 2), `_creditAuthorizationType` will hold 2. If the operator then tries to revoke the authorization for Service A, the `approveAuthorizationRevocation()` function will incorrectly use `authType 2` (from Service B) to build the revocation call, which will likely fail or lead to unintended consequences. The authorization `authType` for Service A is lost.

Recommendation(s): Consider redesigning the authorization mechanism to correctly support one or multiple services. If the intent is to support borrowing from multiple services, the authorization type should be stored on a per-service basis (e.g., mapping(address => IService.AuthorizationType)). If the intent is to limit authorization to only one service at a time, consider adding checks to `authorizeOperatorBorrowing()` to revert if an authorization is already active.

Status: Fixed

Update from the client: fix commit a787551

6.3 [Low] Retroactive Fee Application in Withdrawal Processing

File(s): ManagementAccount.sol

Description: When a user calls `processWithdrawal`, the withdrawal fee is read from the factory contract at processing time, not at the time the withdrawal was requested. This means that if the factory admin updates the withdrawal fee between when a user requests a withdrawal and when it is processed, the user will be charged the new fee rate instead of the fee rate that was in effect when they initiated the withdrawal.

```
function processWithdrawal(
        uint256 requestId,
        mapping(uint256 => IManagementAccount.WithdrawalRequest) storage withdrawalRequests,
        mapping(uint256 => bool) storage withdrawalApprovals,
        mapping(address => uint256) storage lockedBalances,
        address settlementToken,
        address factory

    )

        internal
    {
       //...
        // .... rest of the code
       //@audit retroactive fees application given the             withdrawal delay.
        IManagementAccountFactory factoryContract = IManagementAccountFactory(factory);
        uint256 withdrawalFee = factoryContract.withdrawalFee();
        address feeRecipient = factoryContract.feeRecipient();
        // ... rest of the code
        // ...
        // ...
```

Recommendation(s): Store the withdrawal fee rate in the `WithdrawalRequest` struct at the time of request creation. The `WithdrawalRequest` struct should include a `feeRate` field that captures the fee at request time:

Status: Fixed

Update from the client: fix commit e2446be3

6.4 [Low] Approvals granted in credit mode are not revoked when switching to spending mode

File(s): ManagementAccount.sol

Description: The ManagementAccount contract operates using different modes, such as CREDIT and SPENDING. In CREDIT mode, the operator is granted specific approvals to borrow funds from a configured credit service on behalf of the owner.

The executeModeChange() function is responsible for transitioning the account between these modes after a cooldown period. When the account is switched to SPENDING mode, the function correctly checks if the available settlement token balance meets the required minimum.

```
function executeModeChange() external onlyOperator {
    if (_pendingModeChangeTime == 0) {
        revert ManagementAccountErrors.NoModeChangePending();
    }
    if (block.timestamp < _pendingModeChangeTime) {
        revert ManagementAccountErrors.ModeChangeInCooldown(
            _pendingModeChangeTime,
            uint64(block.timestamp)
        );
    }

    // @audit Enforces settlement token minimum.
    if (_pendingMode == AccountMode.SPENDING && _spendingLimit.settlementAllowance > 0) {
        address settlementTokenAddress = _settlementToken();
        uint26 availableBalance = _availableBalance(settlementTokenAddress);
        if (availableBalance < _spendingLimit.settlementAllowance) {
            revert ManagementAccountErrors.SettlementBalanceBelowSpendingLimit(
                availableBalance,
                _spendingLimit.settlementAllowance
            );
        }
    }

    // @audit-issue Fails to revoke approvals granted to the credit service.

    _mode = _pendingMode;
    _pendingMode = AccountMode.NONE;
    _pendingModeChangeTime = 0;

    emit ModeChanged(_mode);
}
```

However, the function does not include logic to revoke the approvals that were previously granted to the credit service when the account was in CREDIT mode.

This means that even after the account is switched to SPENDING mode, the credit service (or the operator) will still hold valid approvals to borrow on behalf of the ManagementAccount. This violates the intended separation of permissions between modes and could allow the operator to perform credit-related actions, such as borrowing, when the account is explicitly intended to be in SPENDING mode.

Recommendation(s): Consider updating the executeModeChange() function so that when the mode is switched from CREDIT to SPENDING, all outstanding approvals related to the credit service are properly revoked.

Status: Acknowledged

Update from the client: Acknowledged

6.5 [Low] Extended Wait Time for Settlement Token Withdrawals Due to Sequential Approval Process

File(s): ManagementAccount.sol

Description: For settlement token withdrawals in SPENDING mode, users must wait for both the cooldown period AND operator approval before they can process the withdrawal. The `approveWithdrawal` function enforces that the cooldown period must have elapsed before approval can be granted, and `processWithdrawal` requires approval to be set. This creates a sequential dependency where:

1. User requests withdrawal → cooldown period starts ;
2. Cooldown period ends → operator can approve ;
3. Operator approves → user can process withdrawal ;

This means users effectively wait: `cooldown period + operator response time`, rather than just the cooldown period.

```
// In ManagementAccount.approveWithdrawal(...)
if (block.timestamp < request.executableAt) {
    revert ManagementAccountErrors.WithdrawalInCooldown(...);
}

// In ManagementAccountLib.processWithdrawal(...)
if (isSettlementInSpending) {
    if (!withdrawalApprovals[requestId]) {
        revert ManagementAccountErrors.CollateralWithdrawalNotApproved(requestId);
    }
}
```

Recommendation(s): Consider properly documenting the design if this is intentional, and users understand the two-step process and expect wait times. Otherwise, you can consider allowing the operator to approve before the cooldown period has elapsed or factor in the operator approval period within the cooldown duration.

Status: Acknowledged

Update from the client: It's intentional and will be clear for users

6.6 [Low] Fee calculations round in favor of users instead of the protocol

File(s): `ManagementAccount.sol`, `ManagementAccountLib.sol`

Description: The withdrawal fee calculation in both `_executeInstantWithdrawal(...)` and `processWithdrawal(...)` functions uses integer division to calculate the fee amount. When calculating fees using `(amount * withdrawalFee) / 10_000`, any remainder from the division is effectively rounded down, which means the protocol receives slightly less than the intended fee percentage. This rounding error benefits users at the expense of the protocol, and over many transactions, these small rounding errors can accumulate into a significant amount.

Recommendation(s): Consider implementing a rounding mechanism that rounds fees in favour of the protocol. This can be achieved by adding the divisor minus one before dividing (e.g., `(amount * withdrawalFee + 9999) / 10_000`). This ensures that any rounding errors benefit the protocol rather than users.

Status: Acknowledged

Update from the client: Acknowledgded

6.7 [Low] ManagementAccount contract does not support native tokens

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract is designed to execute arbitrary service actions. However, the contract itself cannot receive or hold native tokens, as it does not implement a `receive()` or `fallback()` function, nor does it contain any `payable` functions.

Despite this, two functions, `executeApprovedServiceAction()` and `approveAuthorizationRevocation()` utilize low-level calls that are configured to forward a `value` (call.value).

The `executeApprovedServiceAction()` function forwards `call.value` when executing a service action:

```
function executeApprovedServiceAction(...)
    external
    nonReentrant
    returns (bytes[] memory results)
{
    // ...
    for (uint256 i; i < length; ++i) {
        IService.Call memory call = calls[i];
        // @audit-issue This call forwards `call.value`.
        (bool success, bytes memory response) = call.target.call{ value: call.value }(
            call.callData
        );
        if (!success) ManagementAccountLib.revertWithReason(response);
        // ...
    }
}
```

Similarly, the `approveAuthorizationRevocation()` function also forwards `call.value`.

```
function approveAuthorizationRevocation() external onlyOperator nonReentrant {
    // ...
    uint256 length = calls.length;
    for (uint256 i; i < length; ++i) {
        IService.Call memory call = calls[i];
        // @audit-issue This call also forwards `call.value`.
        (bool success, ) = call.target.call{ value: call.value }(call.callData);
        // ...
    }
}
```

If any configured service attempts to execute a call where `call.value` is greater than 0, the low-level call will attempt to pull native tokens from the `ManagementAccount contract`. Since the contract cannot hold a native token balance, this attempt will fail, resulting in the entire transaction being reverted. This creates a denial-of-service (DoS) condition for any service actions that legitimately require sending native tokens, or if an attacker can configure a service to do so.

Recommendation(s): Consider reconciling this contradictory logic. If the `ManagementAccount` contract is intended to support native token transfers, it must be made payable (e.g., by adding a' receive() ' function) so that it can receive and hold a balance. If it is not intended to support native tokens, then the logic in both functions should be updated to explicitly revert or require(call.value == 0, "Native token transfers not supported")

Status: Acknowledged

Update from the client: The smart account by design doesn't support native tokens

6.8 [Low] Old Operator Authorization Not Revoked When Operator Changes

File(s): ManagementAccount.sol

Description: The `revokeOperatorBorrowing` function reads the current operator address from the factory contract. If the factory admin updates the operator address before `revokeOperatorBorrowing` is called, the function will attempt to revoke authorization for the new operator instead of the old operator. This leaves the old operator with active authorization.

```
function revokeOperatorBorrowing(
    address service,
    IService.AuthorizationType authType,
    address authorized
)
    external
    override
    onlyOwner
    nonReentrant
    returns (bytes[] memory results)
{
    address operatorAddress = IManagementAccountFactory(_factory).operator(); //@audit allow revocation of old operators

    // Prevent direct revocation of OPERATOR's credit service authorization in CREDIT mode
    if (_mode == AccountMode.CREDIT && service == _creditService && authorized == operatorAddress) {
        revert ManagementAccountErrors.CannotDirectlyRevokeCreditServiceAuth(service);
    }

    // Uses the 'authorized' parameter, but the check above uses operatorAddress from factory
    IService.Call[] memory calls =
        IService(service).buildRevokeAuthorization(authType, authorized, settlementTokenAddress);
    results = _executeServiceCalls(service, calls);
}
```

Recommendation(s): Consider revoking old operator authorization in all accounts once updated or store the operator address and revoke against he stored address.

Status: Acknowledged

Update from the client: Acknowledged

6.9 [Low] Settlement token in ManagementAccountFactory can be inconsistent with TokenWhitelistRegistry

File(s): ManagementAccountFactory.sol, TokenWhitelistRegistry

Description: The `ManagementAccountFactory` contract is configured with a `settlementToken` address during initialization and can be updated later by an admin with the `IMPLEMENTATION_ADMIN_ROLE`. This `settlementToken` is intended to be consistent with the settlement token specified in the `TokenWhitelistRegistry` contract.

However, the factory contract allows the `settlementToken` to be set to any arbitrary address. The `initialize()` function takes it as a parameter without validation:

```
 function initialize(
    address admin,
    address implementation_,
    address registry_,
    address tokenWhitelistRegistry_,
    address settlementToken_,
    // ...
) external initializer {
    // ...
    tokenWhitelistRegistry = tokenWhitelistRegistry_;
    // @audit-issue settlementToken_ is set directly without validation.
    settlementToken = settlementToken_;
    // ...
}
```

Similarly, the `setSettlementToken()` function also allows the admin to set any non-zero address, without checking against the `TokenWhitelistRegistry`.

```
function setSettlementToken(address newToken) external onlyRole(IMPLEMENTATION_ADMIN_ROLE) {
    require(newToken != address(0), "Factory: settlement token is zero");
    address previous = settlementToken;
    // @audit-issue newToken is set without validating against TokenWhitelistRegistry.
    settlementToken = newToken;
    emit SettlementTokenUpdated(previous, newToken);
}
```

This creates a risk of misconfiguration where the `ManagementAccountFactory` operates with a `settlementToken` that is different from the one defined in the `TokenWhitelistRegistry`. Such a discrepancy could lead to system inconsistencies or failed transactions if other components rely on these two values being synchronized.

Recommendation(s): Consider enforcing consistency by removing the ability to set the `settlementToken` manually. Instead, `ManagementAccountFactory` could fetch the correct token address directly from the `TokenWhitelistRegistry` contract. If the `TokenWhitelistRegistry` address itself can be updated, the logic for that update should also include automatically refreshing the `settlementToken` in the same transaction.

Status: Fixed

Update from the client:fix commit d527b1446a32711e36b335346167fd5f8b1a15d1. The settlement token logic has been removed from TokenWhitelistRegistry to eliminate storage divergence risk:

- TokenWhitelistRegistry - Now a pure token whitelist without special settlement token handling
- ManagementAccountFactory - Manages settlement token references independently
- Settlement Token Whitelisting - Now manual (admins must explicitly call tokenRegistry.whitelistToken())
- Settlement Token Removal - Settlement tokens can now be removed from the registry like any other token

6.10 [Low] The `ManagementAccount` contract uses `transfer` and `transferFrom` which may fail for non-standard ERC20 tokens

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract utilizes the standard `transfer` and `transferFrom` functions defined in the IERC20 interface to move tokens.

However, not all ERC20 tokens adhere strictly to the standard. Some widely used tokens, such as `USDT`, do not return a boolean value upon a successful transfer (they return void). Others might return false instead of reverting on failure.

When the contract is compiled with an interface that expects a boolean return value (which is the standard behavior), calling `transfer` or `transferFrom` on a token like `USDT` will cause the transaction to revert because the EVM expects a return value that is not provided.

```
function _executeTransfer(address token, address to, uint256 amount) internal {
    // ...
    // @audit-issue This will revert for tokens that do not return a bool (e.g. USDT)
    IERC20(token).transfer(to, amount);
    // ...
}
```

This limits the protocol's compatibility with a significant portion of the ERC20 ecosystem, potentially leading to stuck funds or inoperable functionality for specific assets.

Recommendation(s): Consider using the `SafeERC20` library from `OpenZeppelin`. This library handles non-standard ERC20 tokens correctly by verifying the return data and ensuring the transfer was successful, regardless of whether the token returns a boolean or no value.

Status: Fixed

Update from the client: fix commit 8489657e084a0de2d32a095b5a3c18c426d2baa0

6.11 [Low] The `MorphoService` approves tokens without zeroing allowance first causing reverts for USDT-like tokens

File(s): MorphoService.sol

Description: The `MorphoService` contract facilitates interactions with the Morpho protocol, requiring the contract to approve the underlying assets so that Morpho can pull tokens for deposits or repayments.

However, the current implementation in `buildDeposit()` and `buildRepay()` functions sets the token allowance directly to the desired amount. It does not check if the current allowance is non-zero, nor does it reset the allowance to zero before setting a new value.

Some non-standard ERC20 tokens (most notably USDT on Ethereum) revert when trying to change an allowance from a non-zero value to another non-zero value. This creates a potential denial-of-service vulnerability. If the `MorphoService` contract already has a residual allowance for the morpho address (e.g., from a previous transaction that didn't consume the full amount), any subsequent attempt to call `buildDeposit()` or `buildRepay()` will revert.

The `buildDeposit()` function constructs the approval call as seen below:

```
// ...
calls[0] = Call({
    target: asset,
    // @audit-issue This line approves a specific amount without resetting allowance to zero.
    callData: abi.encodeWithSignature("approve(address,uint256)", address(morpho), amount),
    value: 0
});
// ...
```

Similarly, the `buildRepay()` function exhibits the same behavior:

```
// ...
calls[0] = Call({
    target: asset,
    // @audit-issue This line approves a specific amount without resetting allowance to zero.
    callData: abi.encodeWithSignature("approve(address,uint256)", address(morpho), amount),
    value: 0
});
// ...
```

Recommendation(s): Consider using OpenZeppelin's SafeERC20 library, specifically the `forceApprove()` function, which handles the non-standard approval logic automatically.

Status: Fixed

Update from the client: 140e887

6.12 [Low] The `executeModeChange()` function fails to revoke approvals when switching credit services

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract allows the owner to switch the underlying credit service provider while the account is in `CREDIT` mode. This is a two-step process involving the `requestCreditModeChange()` function, initiated by the `owner`, and the `executeModeChange()` function, triggered by the `operator` after a delay.

The `executeModeChange()` function is responsible for finalizing the state transition. When switching between services within `CREDIT` mode, it updates the `_creditService` state variable to the new address. However, it fails to revoke any existing token approvals or permissions granted to the previous service provider.

The following code snippet shows how the change is requested:

```
function requestCreditModeChange(address newService) external onlyOwner {
    // ...
    // @audit The new service is staged here
    _pendingCreditService = newService;

    emit CreditModeChangeRequested(newService, executableAt);
}
```

The issue manifests in the execution function below:

```
function executeModeChange() external onlyOperator {
    // ...

    // Apply pending credit config when switching TO CREDIT mode (from SPENDING or within CREDIT)
    if (_pendingMode == AccountMode.CREDIT && _pendingCreditService != address(0)) {
        // @audit-issue The _creditService is updated, but approvals for the old service are not revoked.
        _creditService = _pendingCreditService;

        // Always emit CreditModeChangeExecuted to notify about credit service being set
        emit CreditModeChangeExecuted(_creditService);

        // Clear pending credit change
        _pendingCreditService = address(0);
    }
    // ...
}
```

Because the previous service is simply overwritten, any ERC20 allowances previously granted to it remain active. This violates the principle of least privilege and leaves the account vulnerable if the previous service is compromised or acts maliciously after being replaced.

Recommendation(s): Consider implementing a mechanism to revoke approvals for the current _creditService before the new service requested is executed.

Status: Acknowledged

Update from the client: Acknowledged

6.13 [Info] `registerMarket` in MorphoService allows registering non-existent Morpho Blue markets

File(s): MorphoService.sol

Description: The `MorphoService` contract is responsible for interacting with the Morpho Blue protocol, which is abstracted via the `IMorpho` interface. The service includes a `registerMarket()` function that allows an operator to register new markets for the service to interact with.

```
function registerMarket(MarketParams memory params) external onlyOwner {
        Id marketId = params.id();
        require(!isMarketRegistered[marketId], "MorphoService: already registered");
        require(params.collateralToken != address(0), "MorphoService: zero collateral");
        require(params.loanToken != address(0), "MorphoService: zero loan token");

        marketParamsById[marketId] = params;
        isMarketRegistered[marketId] = true;
        collateralMarket[params.collateralToken] = marketId;
        loanMarket[params.loanToken] = marketId;

        // Does not interact with Morpho to check for existance of market or create if not existing

        emit MarketRegistered(marketId, params.collateralToken, params);
    }
```

However, the current implementation of `registerMarket()` only stores the provided market parameters in the contract's local storage. It does not perform any checks to ensure that this market actually exists on the Morpho Blue protocol, nor does it attempt to create the market by calling the `createMarket()` function available in the IMorpho interface.

```
// function in Morpho Info
function createMarket(MarketParams memory marketParams) external;
```

This creates a discrepancy where the `MorphoService` can have a market registered internally that does not exist externally on Morpho Blue. If any subsequent operations (e.g., supply, borrow, withdraw) attempt to use this registered market, the underlying calls to the Morpho Blue protocol will fail, resulting in the transactions being reverted. This can lead to a denial-of-service condition for parts of the system that rely on this registered market.

As a result, if `registerMarket` function registers a market that does not exist in the Morpho protocol, it will result in reverts.

The `registerMarket` of `MorphoService` should either register new markets if not existing or prevent storing non-existing markets in its local storage.

Recommendation(s): Consider updating the logic within `registerMarket()`. One approach is to prevent the registration of non-existent markets. This can be done by calling the `idToMarketParams()` function on the IMorpho interface to verify the market exists before saving it to storage. If the market does not exist, the transaction should revert.

Alternatively, if the `MorphoService` is intended to have market creation permissions, consider having `registerMarket()` first check for the market's existence and then call `createMarket()` on the `IMorpho` interface if it does not yet exist.

Status: Acknowledged

Update from the client: Acknoweldged as owner of the service we register only existing markets and after DD

6.14 [Info] Approved service actions can execute without re-checking registry or owner approval status

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract is designed to enforce strict invariants regarding service actions. Specifically, it requires that any service attempting to execute an action must be registered, active, and explicitly approved by the owner. While the `executeServiceAction()` function enforces these checks during immediate execution, the deferred execution path fails to do so.

The `executeApprovedServiceAction()` function is used to finalize and execute actions that were previously queued. However, this function relies solely on the state saved within the `PendingServiceAction` struct. It does not re-validate whether the service is still active in the registry or if the owner has maintained their approval.

This omission creates a race condition or state staleness issue. If a service is compromised and is subsequently removed from the registry, a pending action associated with that service can still be executed. This allows a deactivated service to perform operations on the account, violating the security invariant.

```
function executeApprovedServiceAction(uint256 actionId, string calldata action)
    external
    nonReentrant
    returns (bytes[] memory results)
{
    PendingServiceAction storage pending = _pendingServiceActions[actionId];
    if (pending.createdAt == 0) revert ManagementAccountErrors.ServiceActionNotFound(actionId);

    if (!pending.approved) {
        revert ManagementAccountErrors.ServiceActionNotApproved(actionId);
    }

    bytes32 actionHash = keccak256(bytes(action));
    if (actionHash != pending.actionHash) {
        revert ManagementAccountErrors.ServiceActionMismatch(actionId);
    }

    // @audit-issue The function executes calls without re-checking registry status or owner approval.
    // ... calls = IService(pending.service).buildWithdraw(...);
    // ...
    delete _pendingServiceActions[actionId];
}
```

Recommendation(s): Consider modifying the execution flow to verify the current status of the service before finalizing the action. Ensure that the service is still registered, active, and approved by the owner at the moment `executeApprovedServiceAction()` is called.

Status: Acknowledged

Update from the client: Acknowledged

6.15 [Info] Approved service actions cannot be rescinded and never expire

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract enables operators to approve service actions, which subsequently enter a pending state waiting for execution. However, the current logic explicitly prevents the contract owner from rescinding or cancelling a service action once it has been marked as approved.

The `cancelServiceAction()` function includes a check that reverts the transaction if the action's approved status is true. This creates a scenario where an approved action becomes permanently executable. Furthermore, the system lacks an expiration mechanism for these approvals.

```
function cancelServiceAction(uint256 actionId) external onlyOwner {
    PendingServiceAction storage pending = _pendingServiceActions[actionId];
    if (pending.createdAt == 0) {
        revert ManagementAccountErrors.ServiceActionNotFound(actionId);
    }
    // @audit-issue The owner cannot cancel the action if it was already approved.
    if (pending.approved) {
        revert ManagementAccountErrors.ServiceActionAlreadyApproved(actionId);
    }
    delete _pendingServiceActions[actionId];
    emit ServiceActionCancelled(actionId);
}
```

If the owner changes their mind or if market conditions shift, there is no provision to cancel.

Recommendation(s): Consider removing the validation check that prevents cancelling approved actions to allow the owner to rescind permissions if desired.

Status: Acknowledged

Update from the client: Acknowledged

6.16 [Info] Misleading error name when processing an already cancelled withdrawal

File(s): `ManagementAccountLib.sol`

Description: The `processWithdrawal(...)` function in `ManagementAccountLib` checks if a withdrawal request has been cancelled and reverts with `WithdrawalNotCancellable` error. However, this error name is misleading because it suggests that the withdrawal cannot be cancelled, when in fact the withdrawal has already been cancelled. The error message doesn't accurately describe the actual state of the withdrawal request, which can confuse developers and users when debugging issues.

Recommendation(s): Consider using a more descriptive error name, such as `WithdrawalAlreadyCancelled` or `WithdrawalCancelled`, to accurately reflect that the withdrawal request has already been cancelled and cannot be processed.

Status: Acknowledged

Update from the client: Acknowledged

6.17 [Info] The `_pendingMode` state variable is not cleared after execution or cancellation

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract utilizes the `_pendingMode` state variable to store the requested account mode during the mandatory delay period. The validity of this request is tracked via the `_pendingModeChangeTime` timestamp.

However, when a request is either finalized via `executeModeChange()` or discarded via `cancelModeChange()`, the contract only resets the timestamp. The `_pendingMode` variable is left holding the stale value of the previously requested mode.

```
function executeModeChange() external onlyOperator {
    // ...
    _mode = _pendingMode;
    _pendingModeChangeTime = 0;

    // @audit-issue _pendingMode is not cleared and retains the value of the new mode
    emit AccountModeUpdated(_pendingMode);
}


function cancelModeChange() external onlyOwner {
    // ...
    _pendingModeChangeTime = 0;
    // Clear pending credit change if any
    _pendingCreditService = address(0);

    // @audit-issue _pendingMode is not cleared here
    emit ModeChangeCancelled();
}
```

While the logic correctly relies on `_pendingModeChangeTime` to determine if a request is active, leaving stale data in `_pendingMode` results in a "dirty" state. This can lead to confusion for off-chain indexers or developers reviewing the contract state.

Recommendation(s): Consider resetting `_pendingMode` to the current `_mode` or a specific `NONE` value whenever `_pendingModeChangeTime` is reset to 0. This ensures the state clearly reflects that no transition is pending.

Status: Acknowledged

Update from the client: Acknowledged

6.18 [Info] The `cancelModeChange(...)` function clears pending credit service even when it's empty

File(s): `ManagementAccount.sol`

Description: The `cancelModeChange(...)` function allows the owner to cancel a pending mode change request. When cancelling, the function always sets `_pendingCreditService` to `address(0)`, regardless of whether it was previously set. While this operation is safe and doesn't cause any functional issues, it performs an unnecessary storage write when `_pendingCreditService` is already `address(0)`. This results in unnecessary gas consumption when cancelling a mode change that doesn't involve a credit service change.

Recommendation(s): Consider checking if `_pendingCreditService` is non-zero before clearing it. This would save gas when cancelling mode changes that don't involve credit service changes, as it avoids an unnecessary storage write operation.

Status: Acknowledged

Update from the client: Acknowledged

6.19 [Info] The `executeServiceAction()` function bypasses token whitelist validation during the operator approval flow

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract enables owners to perform various service actions, such as withdrawals, via the `executeServiceAction()` function. To ensure protocol security and adherence to risk parameters, the contract restricts these interactions to a whitelist of supported tokens.

However, a logic inconsistency exists when an action requires operator approval. When `executeServiceAction()` determines that approval is needed (e.g., due to credit mode or risk limits), it enqueues the action and returns early. The issue is that the token whitelist validation logic is located after this approval check and early return. Consequently, the validation is skipped during the queuing process.

Subsequently, when the operator approves the action via `executeApprovedServiceAction()`, the function retrieves the stored parameters and executes the logic (building the withdrawal call) without performing the whitelist check that was skipped earlier.

This means that an owner can bypass the whitelist restriction by submitting a request that triggers the approval path. Once approved, they can interact with non-whitelisted tokens, violating the protocol's invariant regarding supported assets.

Recommendation(s): Consider ensuring that token whitelist validation is performed regardless of whether the action requires approval or executes immediately. This can be achieved by adding a validation check inside `executeApprovedServiceAction()` before execution.

Status: Acknowledged

Update from the client: Acknowledged

6.20 [Info] The `executeServiceCalls()` function in `ManagementAccountLib` contains redundant parameters

File(s): `ManagementAccountLib.sol`

Description: The `ManagementAccountLib` library contains the `executeServiceCalls()` function, which is responsible for executing a batch of calls returned by a service. This function accepts `service` and `creditService` as input parameters.

However, these parameters are not utilized within the function's logic. The actual execution relies entirely on the calls array, where each `IService.Call` struct already contains the necessary `target` address and `callData`.

```
function executeServiceCalls(
    // @audit-issue This parameter is unused
    address service,
    // @audit-issue This parameter is unused
    address creditService,
    IService.Call[] memory calls
) internal returns (bytes[] memory results) {
    uint256 length = calls.length;
    results = new bytes[](length);

    for (uint256 i; i < length; ++i) {
        IService.Call memory call = calls[i];
        // @audit Execution uses call.target, ignoring the service params
        (bool success, bytes memory result) = call.target.call{ value: call.value }(call.callData);
        // ... handle success/failure ...
        results[i] = result;
    }
}
```

Passing unused arguments increases the function signature complexity without providing any functional value.

Recommendation(s): Consider removing the `service` and `creditService` parameters from the `executeServiceCalls()` function signature to clean up the code

Status: Acknowledged

Update from the client: Acknowledged

6.21 [Info] The `getAvailableBalance()` function duplicates the logic of `_availableBalance()`

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract contains logic to calculate the spendable balance of a token by subtracting locked amounts from the contract's balance. This logic is implemented in the internal helper `_availableBalance()`.

However, the external view function `getAvailableBalance()` completely re-implements this logic instead of calling the internal helper.

```
function _availableBalance(address token) internal view returns (uint256) {
    uint256 balance = IERC20(token).balanceOf(address(this));
    uint256 locked = _lockedBalances[token];
    return balance > locked ? balance - locked : 0;
}
```

The external function contains the exact same code:

```
function getAvailableBalance(address token) external view returns (uint256 balance) {
    uint256 rawBalance = IERC20(token).balanceOf(address(this));
    uint256 locked = _lockedBalances[token];
    // @audit-issue This logic is a duplicate of _availableBalance(...)
    return rawBalance > locked ? rawBalance - locked : 0;
}
```

This code duplication increases the deployment bytecode size and creates a maintenance burden where changes to the balance calculation logic must be applied in two places.

Recommendation(s): Consider refactoring `getAvailableBalance()` to call the internal `_availableBalance()` function.

Status: Acknowledged

Update from the client: Acknowledged

6.22 [Info] The `getBalance()` function silently suppresses errors from external services

File(s): ManagementAccount.sol

Description: The `getBalance()` function in the `ManagementAccount` contract calculates the total token balance by aggregating the funds held directly by the account and those held in tracked external services.

To prevent the entire transaction from reverting due to a single malfunctioning service, the interaction is wrapped in a try-catch block. However, the catch block is empty, meaning that any error thrown by a service is silently ignored.

```
function getBalance(address token) external view override returns (uint256 balance) {
    balance = IERC20(token).balanceOf(address(this));

    uint256 length = _trackedServices.length;
    for (uint256 i; i < length; ++i) {
        address service = _trackedServices[i];
        // ...
        // @audit-issue Errors are silently suppressed here
        try IService(service).getBalance(token, address(this)) returns (uint256 serviceBalance) {
            balance += serviceBalance;
        } catch {
            // Emit event incase a service throw error
        }
    }
}
```

If a service fails (e.g., due to a pause or internal error), the function will return a partial balance that is lower than the actual total. This incorrect data is returned without any indication of failure, potentially misleading off-chain components or calling contracts.

Recommendation(s): Consider emiting events in the catch block.

Status: Acknowledged

Update from the client: Acknowledged

6.23 [Info] The `requestSpendingMode()` and `requestCreditMode()` functions allow overwriting pending requests

File(s): ManagementAccount.sol

Description: The `ManagementAccount` contract enforces a mandatory delay when switching account modes. The owner initiates this process by calling `requestSpendingMode()` or `requestCreditMode()`, which sets a timestamp `_pendingModeChangeTime` indicating when the change can be executed.

However, these functions do not check if a request is already pending. If the functions are called again while a request is active, the `_pendingModeChangeTime` is recalculated based on the current block.timestamp.

```
function requestSpendingMode() external onlyOwner {
    if (_mode == AccountMode.SPENDING) {
        revert ManagementAccountErrors.ModeChangeNotRequired(uint8(AccountMode.SPENDING));
    }

    uint64 delay = IManagementAccountFactory(_factory).modeChangeDelay();
    uint64 executableAt = uint64(block.timestamp + delay);
    _pendingMode = AccountMode.SPENDING;

    // @audit-issue Overwriting the variable resets the delay timer
    _pendingModeChangeTime = executableAt;

    emit ModeChangeRequested(AccountMode.SPENDING, executableAt);
}
```

Recommendation(s): Consider adding a check to ensure that _pendingModeChangeTime is 0 before allowing a new request to be submitted.

Status: Acknowledged

Update from the client: Acknowledged

6.24 [Info] The codebase contains unused custom error definitions

File(s): ServiceErrors.sol, ServiceRegistryErrors.sol, ManagementAccountErrors.sol

Description: The codebase defines several custom errors that are never referenced or reverted in the implementation contracts. Keeping unused code adds unnecessary noise and can lead to confusion regarding the contract's intended behavior and failure modes.

The following errors appear to be unused:

```
// src/libraries/ServiceErrors.sol
// @audit All errors in this file appear to be unused.

// src/libraries/ServiceRegistryErrors.sol
error InvalidServiceType();

// src/libraries/ManagementAccountErrors.sol
error InvalidAccountMode();
error CannotApproveWithdrawalWithActiveDebt();
error CannotSwitchModeWithActiveDebt();
error CannotUseSettlementAsCollateral();
error SettlementBalanceReduced();
error CollateralWithdrawalRequiresTwoStepProcess();
error CollateralPositionReduced();
error ServicePositionCheckFailed();
```

Recommendation(s): Consider removing the unused error definitions to improve code cleanliness and maintainability.

Status: Acknowledged

Update from the client: Acknowledged

6.25 [Info] The isServiceActive(...) and isServiceValid(...) functions in ServiceRegistry are identical

File(s): ServiceRegistry.sol

Description: The `ServiceRegistry` contract allows external callers to verify the status of a service. It provides two view functions, `isServiceActive()` and `isServiceValid(...)`, for this purpose.

However, both functions contain the exact same implementation logic. They both check if the service exists in the registry and if its active flag is set to true.

```
  function isServiceValid(address service) external view override returns (bool) {
    // @audit-issue This logic is identical to isServiceActive below
    return _serviceExists[service] && _serviceInfo[service].active;
}

function isServiceActive(address service) external view override returns (bool) {
    return _serviceExists[service] && _serviceInfo[service].active;
}
```

This code duplication increases the contract's bytecode size unnecessarily and adds maintenance overhead without providing distinct functionality.

Recommendation(s): Consider removing one of the functions to reduce code duplication

Status: Acknowledged

Update from the client: Acknowledged

6.26 [Info] `getBalance` function in MorphoService returns an incorrect value

File(s): MorphoService.sol, ManagementAccount.sol

Description: The `ManagementAccount` contract's `getBalance()` function is designed to provide a consolidated view of an asset balance. It does this by summing its own local balance with the balances reported by all `_trackedServices`.

```
function getBalance(address token) external view override returns (uint256 balance) {
    balance = IERC20(token).balanceOf(address(this));

    uint256 length = _trackedServices.length;
    for (uint256 i; i < length; ++i) {
        address service = _trackedServices[i];
        if (service == address(0)) continue;
        if (!_approvedServices[service]) continue;
        // @audit Calls IService.getBalance() on configured services.
        try IService(service).getBalance(token, address(this)) returns (
            uint256 serviceBalance
        ) {
            balance += serviceBalance;
        } catch { }
    }
}
```

The `MorphoService` contract, which is one of these services, implements `getBalance()` to report the account's position in Morpho Blue markets.

The below getBalance function reads the total balance of the asset in the Morpho Blue protocol. It first reads the collateral market position to capture the collateral provided by the account. If there is a position in the `loanMarket`, the logic also reads the assets supplied to the market by converting the `supplyShares`.

```
function getBalance(
    address asset,
    address account
) external view override returns (uint256 balance) {
    // Check collateral markets
    Id marketId = collateralMarket[asset];
    if (isMarketRegistered[marketId]) {
        Position memory pos = morpho.position(marketId, account);
        balance += pos.collateral;
    }

    // Also check loan markets where asset is supplied
    Id loanMarketId = loanMarket[asset];
    if (isMarketRegistered[loanMarketId]) {
        Position memory loanPos = morpho.position(loanMarketId, account);
        // @audit Accounts for supplied shares.
        Market memory m = morpho.market(loanMarketId);
        if (m.totalSupplyShares > 0) {
            balance += (
                (uint256(loanPos.supplyShares) * uint256(m.totalSupplyAssets)) /
                    uint256(m.totalSupplyShares)
            );
        }
        // @audit-issue Does not account for `loanPos.borrowShares`.
    }
}
```

This implementation is flawed. It correctly adds assets held as `collateral` (pos.collateral) and assets supplied to a loan market (by converting supplyShares to an asset amount). However, it completely ignores any `borrowShares` the account may have in that same token.

The Morpho Blue Position struct tracks supplied, borrowed, and collateralized assets separately

This is based on how the position fields are tracked separately in Morpho.

```
struct Position {
    uint256 supplyShares;
    uint128 borrowShares;
    uint128 collateral;
}
```

By ignoring `borrowShares`, the `getBalance()` function returns a gross balance (total supplied + collateral) rather than a net balance. For example, if an account has supplied 100 WETH and borrowed 30 WETH from the same market, `getBalance()` will incorrectly report 100 WETH instead of the net position of 70 WETH. This provides a misleading and inflated value of the account's holdings to the ManagementAccount, which could lead to flawed accounting or incorrect decision-making in any higher-level logic that relies on this balance.

IMorpho.sol

Recommendation(s): Consider adding a check for _pendingModeChangeTime to be 0 when accepting the request.

Status: Acknowledged

Update from the client: Acknowledged

6.27 [Best Practices] Inconsistent naming convention in contracts

File(s): ManagementAccountFactory.sol, ServiceRegistry.sol

Description: The codebase exhibits inconsistent naming conventions for state variables across different contracts.

In the ManagementAccountFactory contract, state variables are defined without an underscore prefix, regardless of visibility.

```
contract ManagementAccountFactory is Initializable, AccessControl, UUPSUpgradeable {
    bytes32 public constant IMPLEMENTATION_ADMIN_ROLE = keccak256("IMPLEMENTATION_ADMIN_ROLE");

    // @audit State variables defined without underscore prefix
    address public implementation;
    address public registry;
    address public tokenWhitelistRegistry;
    address public settlementToken;
    address public operator; // Single operator for all accounts
    uint64 public modeChangeDelay;
    // ...
}
```

Conversely, the ServiceRegistry contract follows a different convention, utilizing the underscore prefix for private state variables.

```
contract ServiceRegistry is IServiceRegistry, AccessControl {
    using ServiceRegistryErrors for *;

    bytes32 public constant override SERVICE_ADMIN_ROLE = keccak256("SERVICE_ADMIN_ROLE");

    // @audit State variables defined with underscore prefix
    mapping(address => ServiceInfo) private _serviceInfo;
    mapping(address => bool) private _serviceExists;
    // ...
}
```

Maintaining a consistent coding style and naming convention throughout the project is best practice as it improves code readability and maintainability.

Recommendation(s): Consider adopting a unified naming convention across the entire codebase.

Status: Acknowledged

Update from the client: Acknowledged

6.28 [Best Practices] The `ManagementAccountFactory` contract does not inherit from `IManagementAccountFactory`

File(s): ManagementAccountFactory.sol

Description: It is a standard Solidity practice to define an interface that outlines the required external functions and events for a contract, and then explicitly inherit that interface in the implementation contract. This ensures that the contract adheres to the expected behavior and function signatures.

The `ManagementAccountFactory` contract implements the factory logic but does not declare inheritance from its corresponding interface, `IManagementAccountFactory`.

```
// @audit-issue The contract fails to inherit from IManagementAccountFactory
contract ManagementAccountFactory is Initializable, AccessControl, UUPSUpgradeable {
    // ...
}
```

Missing the explicit inheritance means the compiler cannot verify that the `ManagementAccountFactory` actually implements all functions defined in `IManagementAccountFactory`. This could lead to inconsistencies between the interface definition and the actual implementation.

Recommendation(s): Consider adding `IManagementAccountFactory` to the inheritance list of the `ManagementAccountFactory` contract.

Status: Acknowledged

Update from the client: Acknowledged

6.29 [Best Practices] The `ManagementAccountFactory` contract inherits from non-upgradeable AccessControl

File(s): ManagementAccountFactory.sol

Description: The `ManagementAccountFactory` is implemented as a UUPS upgradeable contract, indicated by its inheritance of `UUPSUpgradeable` and Initializable. When building upgradeable contracts with OpenZeppelin, it is required to use the Upgradeable variants of the libraries to ensure storage safety.

However, the current implementation inherits from the standard, non-upgradeable AccessControl library.

```
// @audit-issue The contract inherits from the non-upgradeable AccessControl
contract ManagementAccountFactory is Initializable, AccessControl, UUPSUpgradeable {
    // ...
}
```

Recommendation(s): Consider replacing `AccessControl` with `AccessControlUpgradeable`.

Status: Acknowledged

Update from the client: Acknowledged

7 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.

Remarks about HyperBeat's documentation

The **HyperBeat's Liquid Bank Contracts** documentation is primarily provided through the `MANAGEMENT_ACCOUNT_DOCUMENTATION.md` in the repository. The codebase itself has comments facilitating code comprehension and auditability. Additionally, the HyperBeat team was responsive and available for internal sync calls, giving a comprehensive overview of contracts to the Nethermind Security team in the kick-off call.

8 Test Suite Evaluation

8.1 Tests Output

```
Ran 1 test for test/unit/Sanity.t.sol:SanityTest
[PASS] testTrue() (gas: 222)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 157.10µs (24.60µs CPU time)

Ran 4 tests for test/unit/ServiceRegistry.t.sol:ServiceRegistryTest
[PASS] testRegisterRequiresRole() (gas: 13090)
[PASS] testRegisterService() (gas: 84494)
[PASS] testRemoveService() (gas: 67319)
[PASS] testUpdateService() (gas: 108131)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.05ms (283.30µs CPU time)

Ran 17 tests for test/unit/ManagementAccountFactoryUpgrade.t.sol:ManagementAccountFactoryUpgradeTest
[PASS] testCannotReinitializeFactoryAfterUpgrade() (gas: 1885417)
[PASS] testFactoryCanCreateAccountsAfterUpgrade() (gas: 2161898)
[PASS] testFactoryUpgradeAndUseNewFeature() (gas: 1873437)
[PASS] testFactoryUpgradeByAdmin() (gas: 1872824)
[PASS] testFactoryUpgradeEventEmitted() (gas: 1873177)
[PASS] testFactoryUpgradePreservesAdminRole() (gas: 1880828)
[PASS] testFactoryUpgradePreservesImplementation() (gas: 1876590)
[PASS] testFactoryUpgradePreservesModeChangeDelay() (gas: 1876940)
[PASS] testFactoryUpgradePreservesRegistry() (gas: 1877426)
[PASS] testFactoryUpgradePreservesSettlementToken() (gas: 1877025)
[PASS] testFactoryUpgradePreservesTokenWhitelistRegistry() (gas: 1876810)
[PASS] testFactoryUpgradePreservesWhitelistedImplementations() (gas: 6625050)
[PASS] testFactoryUpgradePreservesWithdrawalDelay() (gas: 1877517)
[PASS] testFactoryUpgradeRevertsIfNotAdmin() (gas: 1870211)
[PASS] testFactoryUpgradeRevertsIfNotContract() (gas: 19014)
[PASS] testFactoryUpgradeRevertsIfZeroAddress() (gas: 16750)
[PASS] testMultipleFactoryUpgradesInSequence() (gas: 3710741)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 3.24ms (3.40ms CPU time)

Ran 11 tests for test/unit/ManagementAccountGetBalance.t.sol:ManagementAccountGetBalanceTest
[PASS] testGetBalanceAfterServiceDeposit() (gas: 403237)
[PASS] testGetBalanceHandlesServiceErrors() (gas: 215649)
[PASS] testGetBalanceIgnoresRevokedService() (gas: 232433)
[PASS] testGetBalanceMultipleTokens() (gas: 354962)
[PASS] testGetBalanceWithLargeServiceList() (gas: 439140)
[PASS] testGetBalanceWithMixedServiceStates() (gas: 496937)
[PASS] testGetBalanceWithMultipleServices() (gas: 525438)
[PASS] testGetBalanceWithNoServices() (gas: 81180)
[PASS] testGetBalanceWithOneService() (gas: 248872)
[PASS] testGetBalanceWithZeroAddressService() (gas: 249110)
[PASS] testToken() (gas: 2598)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 5.92ms (3.35ms CPU time)

Ran 18 tests for test/unit/ManagementAccountServiceActions.t.sol:ManagementAccountServiceActionsTest
[PASS] testBorrowAction() (gas: 344679)
[PASS] testBorrowActionEmitsEvent() (gas: 344200)
[PASS] testBorrowActionRequiresApprovedService() (gas: 55904)
[PASS] testBorrowActionRequiresWhitelistedToken() (gas: 162285)
[PASS] testBorrowActionWithWhitelistedToken() (gas: 344718)
[PASS] testBorrowAndRepayWorkflow() (gas: 407618)
[PASS] testCancelServiceAction() (gas: 346424)
[PASS] testCancelServiceActionClearsState() (gas: 349445)
[PASS] testCancelServiceActionOnlyOwner() (gas: 415705)
[PASS] testCancelServiceActionRevertsIfAlreadyApproved() (gas: 418310)
[PASS] testCancelServiceActionRevertsIfNotFound() (gas: 22326)
[PASS] testMultipleServiceActionsInSequence() (gas: 467129)
[PASS] testRepayAction() (gas: 331839)
[PASS] testRepayActionEmitsEvent() (gas: 332083)
[PASS] testRepayActionRequiresApprovedService() (gas: 56498)
[PASS] testRepayActionRequiresWhitelistedToken() (gas: 162902)
[PASS] testRepayActionWithSettlementToken() (gas: 367168)
[PASS] testRepayActionWithWhitelistedToken() (gas: 331680)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 6.31ms (5.39ms CPU time)

Ran 14 tests for test/unit/ManagementAccountLib.t.sol:ManagementAccountLibTest
[PASS] testExtractSelectorDoesNotAffectRestOfData() (gas: 652)
```

```
[PASS] testExtractSelectorFuzzed(bytes4,bytes) (runs: 256, : 1577, ~: 1539)
[PASS] testExtractSelectorWithApproveData() (gas: 790)
[PASS] testExtractSelectorWithBalanceOfData() (gas: 750)
[PASS] testExtractSelectorWithCustomSelector() (gas: 836)
[PASS] testExtractSelectorWithEmptyData() (gas: 1036)
[PASS] testExtractSelectorWithExactly4Bytes() (gas: 869)
[PASS] testExtractSelectorWithLessThan4Bytes() (gas: 2044)
[PASS] testExtractSelectorWithMaxSelector() (gas: 839)
[PASS] testExtractSelectorWithMoreThan4Bytes() (gas: 6308)
[PASS] testExtractSelectorWithSelectorHavingLeadingZeros() (gas: 882)
[PASS] testExtractSelectorWithSelectorHavingTrailingZeros() (gas: 1036)
[PASS] testExtractSelectorWithValidData() (gas: 944)
[PASS] testExtractSelectorWithZeroSelector() (gas: 571)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 9.52ms (9.75ms CPU time)

Ran 29 tests for test/unit/ManagementAccountAdditional.t.sol:ManagementAccountAdditionalTest
[PASS] testApproveServiceAction() (gas: 416720)
[PASS] testApproveServiceActionOnlyOperator() (gas: 414906)
[PASS] testApproveServiceActionRevertsIfAlreadyApproved() (gas: 419901)
[PASS] testApproveServiceActionRevertsIfNotFound() (gas: 32859)
[PASS] testCollateralTokenInstantInCreditMode() (gas: 252480)
[PASS] testCustomActionRevertsIfSettlementBalanceBelowSpendingLimit() (gas: 273273)
[PASS] testCustomActionSucceedsIfSettlementBalanceStaysAboveSpendingLimit() (gas: 273628)
[PASS] testCustomActionThatIncreasesSettlementBalance() (gas: 272332)
[PASS] testCustomActionWithLockedSettlementBalance() (gas: 407633)
[PASS] testCustomActionWithNonSettlementTokenNotAffectedBySpendingLimit() (gas: 315616)
[PASS] testExecuteApprovedServiceAction() (gas: 461664)
[PASS] testExecuteApprovedServiceActionRevertsIfNotApproved() (gas: 418403)
[PASS] testExecuteApprovedServiceActionRevertsIfNotFound() (gas: 26627)
[PASS] testExecuteApprovedServiceActionRevertsOnActionMismatch() (gas: 423642)
[PASS] testExecuteCustomAction() (gas: 168133)
[PASS] testExecuteCustomActionInCreditMode() (gas: 201749)
[PASS] testExecuteCustomActionRequiresApprovedService() (gas: 54428)
[PASS] testExecuteCustomActionWithDifferentData() (gas: 168000)
[PASS] testExecuteCustomActionWithoutSpendingLimit() (gas: 167930)
[PASS] testInstantWithdrawalRevertsOnInsufficientBalance() (gas: 94465)
[PASS] testInstantWithdrawalWithLockedBalance() (gas: 138088)
[PASS] testRequestRevokeServiceAuthorization() (gas: 227825)
[PASS] testRequestRevokeServiceAuthorizationOnlyOwner() (gas: 184494)
[PASS] testRequestRevokeServiceAuthorizationRequiresCreditMode() (gas: 140178)
[PASS] testRequestRevokeServiceAuthorizationRequiresCreditService() (gas: 181174)
[PASS] testRequestRevokeServiceAuthorizationRevertsIfAlreadyPending() (gas: 229670)
[PASS] testSettlementTokenInstantInCreditMode() (gas: 252439)
[PASS] testSettlementTokenRequiresCooldownInSpendingMode() (gas: 244021)
[PASS] testWithdrawalCooldownInUnknownMode() (gas: 106701)
Suite result: ok. 29 passed; 0 failed; 0 skipped; finished in 8.95ms (6.91ms CPU time)

Ran 19 tests for test/unit/ManagementAccountCreditModeChange.t.sol:ManagementAccountCreditModeChangeTest
[PASS] testCancelCreditModeChange() (gas: 54069)
[PASS] testCancelCreditModeChangeRevertsIfNoChange() (gas: 21457)
[PASS] testCancelCreditModeChangeRevertsIfNotOwner() (gas: 72542)
[PASS] testCannotRequestRegularModeChangeWhileCreditChangeIsPending() (gas: 79081)
[PASS] testCreditModeChangeAfterCancellation() (gas: 83624)
[PASS] testCreditModeChangeThenSwitchToSpending() (gas: 71432)
[PASS] testExecuteCreditModeChange() (gas: 60479)
[PASS] testExecuteCreditModeChangeOnlyService() (gas: 58568)
[PASS] testExecuteCreditModeChangeRevertsIfInCooldown() (gas: 71833)
[PASS] testExecuteCreditModeChangeRevertsIfNoChange() (gas: 31468)
[PASS] testExecuteCreditModeChangeRevertsIfNotOperator() (gas: 69820)
[PASS] testMultipleCreditModeChanges() (gas: 90096)
[PASS] testRequestCreditModeChange() (gas: 66710)
[PASS] testRequestCreditModeChangeOnlyService() (gas: 66490)
[PASS] testRequestCreditModeChangeRevertsIfNoChange() (gas: 25816)
[PASS] testRequestCreditModeChangeRevertsIfNotInCreditMode() (gas: 48941)
[PASS] testRequestCreditModeChangeRevertsIfNotOwner() (gas: 21541)
[PASS] testRequestCreditModeChangeRevertsIfServiceNotApproved() (gas: 874937)
[PASS] testRequestCreditModeChangeRevertsIfServiceZeroAddress() (gas: 22234)
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 10.99ms (2.08ms CPU time)

Ran 19 tests for test/unit/ManagementAccountFactory.t.sol:ManagementAccountFactoryTest
[PASS] testAdminCanGrantDeployerRole() (gas: 46018)
[PASS] testAdminCanRevokeDeployerRole() (gas: 38420)
[PASS] testAdminHasDeployerRoleByDefault() (gas: 14370)
[PASS] testCreateAccount() (gas: 324613)
[PASS] testCreateAccountEmitsEvent() (gas: 304359)
```

```
[PASS] testCreateAccountRequiresDeployerRole() (gas: 302048)
[PASS] testCreateAccountRevertsForNonDeployer() (gas: 17342)
[PASS] testGrantedDeployerCanCreateAccount() (gas: 336466)
[PASS] testInitializeRejectsZeroAdmin() (gas: 1897633)
[PASS] testInitializeRejectsZeroImplementation() (gas: 1895435)
[PASS] testInitializeRejectsZeroRegistry() (gas: 1895920)
[PASS] testInitializeRejectsZeroSettlementToken() (gas: 1896202)
[PASS] testInitializeRejectsZeroTokenWhitelistRegistry() (gas: 1896200)
[PASS] testMultipleDeployersCanCreateAccounts() (gas: 646789)
[PASS] testNonAdminCannotGrantDeployerRole() (gas: 20538)
[PASS] testNonAdminCannotRevokeDeployerRole() (gas: 20830)
[PASS] testRevokedDeployerCannotCreateAccount() (gas: 321181)
[PASS] testSetImplementationRequiresAdmin() (gas: 4769803)
[PASS] testUpdateImplementation() (gas: 5067316)
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 10.92ms (5.18ms CPU time)

Ran 23 tests for test/unit/ManagementAccountUpgrade.t.sol:ManagementAccountUpgradeTest
[PASS] testCannotReinitializeAfterUpgrade() (gas: 4810757)
[PASS] testCannotUpgradeToSameImplementation() (gas: 38752)
[PASS] testMultipleUpgradesInSequence() (gas: 9571450)
[PASS] testUpgradeAndUseNewFeature() (gas: 4807311)
[PASS] testUpgradeByOwner() (gas: 4807423)
[PASS] testUpgradeDoesNotResetNextWithdrawalId() (gas: 5162689)
[PASS] testUpgradeEventEmitted() (gas: 4807441)
[PASS] testUpgradePreservesCreditConfig() (gas: 4961317)
[PASS] testUpgradePreservesMode() (gas: 4961736)
[PASS] testUpgradePreservesOperator() (gas: 4811113)
[PASS] testUpgradePreservesOwner() (gas: 4812483)
[PASS] testUpgradePreservesPendingModeChange() (gas: 4960184)
[PASS] testUpgradePreservesPendingWithdrawal() (gas: 5103795)
[PASS] testUpgradePreservesServiceApprovals() (gas: 4920333)
[PASS] testUpgradePreservesSettlementAllowance() (gas: 4924233)
[PASS] testUpgradePreservesTokenBalance() (gas: 4889496)
[PASS] testUpgradeRevertsIfNotContract() (gas: 29878)
[PASS] testUpgradeRevertsIfNotOwner() (gas: 4798583)
[PASS] testUpgradeRevertsIfNotUpgraderRole() (gas: 4798341)
[PASS] testUpgradeRevertsIfNotWhitelisted() (gas: 4775140)
[PASS] testUpgradeRevertsIfZeroAddress() (gas: 29592)
[PASS] testUpgradeSucceedsWithWhitelistedImplementation() (gas: 4809432)
[PASS] testUpgradeWorksWithLockedBalances() (gas: 5060678)
Suite result: ok. 23 passed; 0 failed; 0 skipped; finished in 8.77ms (11.52ms CPU time)

Ran 11 tests for test/unit/ManagementAccountAuthRevocation.t.sol:ManagementAccountAuthRevocationTest
[PASS] testApproveAuthorizationRevocation() (gas: 422110)
[PASS] testApproveAuthorizationRevocationCallsService() (gas: 456064)
[PASS] testApproveAuthorizationRevocationClearsState() (gas: 455993)
[PASS] testApproveAuthorizationRevocationOnlyOperator() (gas: 495107)
[PASS] testApproveAuthorizationRevocationRevertsIfNoPending() (gas: 189624)
[PASS] testAuthorizationRevocationFullWorkflow() (gas: 508869)
[PASS] testCancelAuthorizationRevocation() (gas: 454931)
[PASS] testCancelAuthorizationRevocationClearsState() (gas: 496653)
[PASS] testCancelAuthorizationRevocationOnlyOwner() (gas: 495555)
[PASS] testCancelAuthorizationRevocationRevertsIfNoPending() (gas: 182777)
[PASS] testMultipleRevocationCycles() (gas: 547595)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 10.92ms (5.91ms CPU time)

Ran 115 tests for test/unit/ManagementAccount.t.sol:ManagementAccountTest
[PASS] testApproveDecreaseSettlementAllowance() (gas: 171921)
[PASS] testApproveDecreaseSettlementAllowanceAlreadyApproved() (gas: 174059)
[PASS] testApproveDecreaseSettlementAllowanceOnlyOperator() (gas: 146450)
[PASS] testApproveDecreaseSettlementAllowanceRequiresPending() (gas: 35953)
[PASS] testApproveService() (gas: 138692)
[PASS] testApproveServiceMustBeActive() (gas: 63150)
[PASS] testApproveServiceOnlyOwner() (gas: 22158)
[PASS] testApproveServiceRequiresRegistryListing() (gas: 806600)
[PASS] testApproveWithdrawal() (gas: 258858)
[PASS] testApproveWithdrawalEmitsEvent() (gas: 256578)
[PASS] testApproveWithdrawalOnlyOperator() (gas: 226961)
[PASS] testApproveWithdrawalRequiresCooldown() (gas: 229714)
[PASS] testApproveWithdrawalRequiresPending() (gas: 259045)
[PASS] testAuthorizeOperatorBorrowing() (gas: 353133)
[PASS] testAuthorizeOperatorBorrowingCallsService() (gas: 351576)
[PASS] testAuthorizeOperatorBorrowingEmitsEvent() (gas: 355509)
[PASS] testAuthorizeOperatorBorrowingOnlyOwner() (gas: 186621)
[PASS] testAuthorizeOperatorBorrowingRequiresCreditMode() (gas: 148321)
```

```
[PASS] testCanRevokeCreditServiceInSpendingMode() (gas: 353275)
[PASS] testCanRevokeNonCreditServiceInCreditMode() (gas: 1255751)
[PASS] testCanRevokeNonOperatorAuthInCreditMode() (gas: 304959)
[PASS] testCancelDecreaseSettlementAllowance() (gas: 151923)
[PASS] testCancelDecreaseSettlementAllowanceAfterApproval() (gas: 136525)
[PASS] testCancelDecreaseSettlementAllowanceOnlyOwner() (gas: 144417)
[PASS] testCancelDecreaseSettlementAllowanceRequiresPending() (gas: 23210)
[PASS] testCancelModeChange() (gas: 178294)
[PASS] testCancelProcessedWithdrawalReverts() (gas: 265133)
[PASS] testCancelWithdrawal() (gas: 212922)
[PASS] testCancelWithdrawalEmitsEvent() (gas: 208703)
[PASS] testCancelWithdrawalOnlyOwner() (gas: 226317)
[PASS] testCancelWithdrawalReleasesLockedFunds() (gas: 217859)
[PASS] testCannotRequestMultipleDecreasesPending() (gas: 142550)
[PASS] testCannotRevokeCreditServiceInCreditMode() (gas: 362321)
[PASS] testCannotRevokeServiceInCreditMode() (gas: 183793)
[PASS] testCreditConfigChangeTimelockEnforcement() (gas: 1128035)
[PASS] testCreditConfigStorage() (gas: 178538)
[PASS] testCreditModeTimelockEnforcement() (gas: 193535)
[PASS] testDecreaseAllowanceAfterSettlement() (gas: 166222)
[PASS] testDecreaseAllowanceThenWithdrawUnlockedFunds() (gas: 330848)
[PASS] testDefaultModeIsSpending() (gas: 14141)
[PASS] testDefaultRoles() (gas: 35751)
[PASS] testDeposit() (gas: 85895)
[PASS] testDepositEmitsEvent() (gas: 79801)
[PASS] testDepositIncreasesBalance() (gas: 95797)
[PASS] testDepositRequiresWhitelistedToken() (gas: 40019)
[PASS] testDepositRestrictedToOwner() (gas: 23517)
[PASS] testDepositSettlementTokenAutoWhitelisted() (gas: 83628)
[PASS] testExecuteDecreaseSettlementAllowance() (gas: 136214)
[PASS] testExecuteDecreaseSettlementAllowanceOnlyOwner() (gas: 177070)
[PASS] testExecuteDecreaseSettlementAllowanceRequiresApproval() (gas: 142898)
[PASS] testExecuteDecreaseSettlementAllowanceRequiresPending() (gas: 24509)
[PASS] testExecuteModeChange() (gas: 184127)
[PASS] testExecuteModeChangeBeforeDelay() (gas: 192966)
[PASS] testExecuteModeChangeNoRequest() (gas: 32569)
[PASS] testExecuteServiceAction() (gas: 368020)
[PASS] testExecuteServiceEmitsEvent() (gas: 369384)
[PASS] testExecuteServiceRequiresActiveListing() (gas: 171280)
[PASS] testExecuteServiceRequiresApproval() (gas: 105565)
[PASS] testFactoryAddressStored() (gas: 15561)
[PASS] testFullDecreaseWorkflow() (gas: 185254)
[PASS] testIncreaseSettlementAllowance() (gas: 113433)
[PASS] testIncreaseSettlementAllowanceNoValidationInCreditMode() (gas: 209634)
[PASS] testIncreaseSettlementAllowanceOnlyOwner() (gas: 20859)
[PASS] testIncreaseSettlementAllowanceValidatesBalanceInSpendingMode() (gas: 127269)
[PASS] testInitialize() (gas: 41285)
[PASS] testInitializeOnlyOnce() (gas: 20554)
[PASS] testInitializeZeroAddress() (gas: 4865087)
[PASS] testModeChangeDelayFactoryControlled() (gas: 13841)
[PASS] testOnlyOperatorFunctions() (gas: 53018)
[PASS] testOnlyOwnerFunctions() (gas: 61754)
[PASS] testOnlyOwnerOrOperatorFunctions() (gas: 270046)
[PASS] testProcessMultipleWithdrawals() (gas: 395049)
[PASS] testProcessWithdrawal() (gas: 270621)
[PASS] testProcessWithdrawalAfterCooldown() (gas: 276826)
[PASS] testProcessWithdrawalEmitsEvent() (gas: 262835)
[PASS] testProcessWithdrawalInvalidId() (gas: 39204)
[PASS] testProcessWithdrawalRequiresApproved() (gas: 235156)
[PASS] testProcessWithdrawalTransfersTokens() (gas: 263168)
[PASS] testProcessWithdrawalUnlocksFunds() (gas: 270338)
[PASS] testRemoveSettlementTokenFromRegistryReverts() (gas: 17766)
[PASS] testRemoveTokenFromRegistry() (gas: 51369)
[PASS] testRequestCreditMode() (gas: 191940)
[PASS] testRequestCreditModeRequiresApprovedService() (gas: 26999)
[PASS] testRequestDecreaseSettlementAllowance() (gas: 141417)
[PASS] testRequestDecreaseSettlementAllowanceOnlyOwner() (gas: 118487)
[PASS] testRequestDecreaseSettlementAllowanceTooLarge() (gas: 116930)
[PASS] testRequestDecreaseSettlementAllowanceZeroAmount() (gas: 118268)
[PASS] testRequestSpendingModeFromCredit() (gas: 175498)
[PASS] testRequestWithdrawal() (gas: 223503)
[PASS] testRequestWithdrawalCollateralInCredit() (gas: 276430)
[PASS] testRequestWithdrawalEmitsEvent() (gas: 224122)
[PASS] testRequestWithdrawalIncrementsRequestId() (gas: 326489)
[PASS] testRequestWithdrawalInsufficientBalance() (gas: 97554)
```

```
[PASS] testRequestWithdrawalInvalidRecipient() (gas: 79050)
[PASS] testRequestWithdrawalLocksFunds() (gas: 225662)
[PASS] testRequestWithdrawalSettlementTokenAllowed() (gas: 220061)
[PASS] testRequestWithdrawalStoresModeAtRequest() (gas: 224493)
[PASS] testRevokeOperatorBorrowing() (gas: 355849)
[PASS] testRevokeOperatorBorrowingCallsService() (gas: 355805)
[PASS] testRevokeService() (gas: 125356)
[PASS] testRoleTransfer() (gas: 55438)
[PASS] testSettlement() (gas: 157673)
[PASS] testSettlementAllowanceTracking() (gas: 172654)
[PASS] testSettlementConsumesAllowance() (gas: 155873)
[PASS] testSettlementEmitsEvent() (gas: 154759)
[PASS] testSettlementExceedingAllowanceReverts() (gas: 126114)
[PASS] testSettlementRequiresOperator() (gas: 117674)
[PASS] testSettlementRequiresSettlementToken() (gas: 167795)
[PASS] testSettlementTokenAlwaysWhitelisted() (gas: 31362)
[PASS] testSettlementWithoutAllowanceReverts() (gas: 97406)
[PASS] testUpgradeFlow() (gas: 4847279)
[PASS] testUpgradeOnlyByUpgrader() (gas: 4744925)
[PASS] testUpgradePreservesStorage() (gas: 4861037)
[PASS] testUpgradeToInvalidImplementation() (gas: 536685)
[PASS] testWhitelistTokenViaRegistry() (gas: 601634)
Suite result: ok. 115 passed; 0 failed; 0 skipped; finished in 8.85ms (28.92ms CPU time)

Ran 45 tests for test/unit/ManagementAccountFactoryAdmin.t.sol:ManagementAccountFactoryAdminTest
[PASS] testAdminOperationsPreserveOtherState() (gas: 31092)
[PASS] testCannotSetRemovedImplementation() (gas: 4753868)
[PASS] testMultipleAdminOperationsInSequence() (gas: 61061)
[PASS] testRemoveImplementationFromWhitelist() (gas: 4757000)
[PASS] testRemoveImplementationFromWhitelistEmitsEvent() (gas: 4753361)
[PASS] testRemoveImplementationFromWhitelistRejectsActiveImplementation() (gas: 23736)
[PASS] testRemoveImplementationFromWhitelistRejectsNotWhitelisted() (gas: 19994)
[PASS] testRemoveImplementationFromWhitelistRejectsZeroAddress() (gas: 17405)
[PASS] testRemoveImplementationFromWhitelistRequiresAdmin() (gas: 4769755)
[PASS] testSetFeeRecipient() (gas: 42030)
[PASS] testSetFeeRecipientEmitsEvent() (gas: 43567)
[PASS] testSetFeeRecipientRejectsZeroAddress() (gas: 17481)
[PASS] testSetFeeRecipientRequiresAdmin() (gas: 17284)
[PASS] testSetModeChangeDelay() (gas: 26865)
[PASS] testSetModeChangeDelayEmitsEvent() (gas: 25143)
[PASS] testSetModeChangeDelayMultipleTimes() (gas: 30171)
[PASS] testSetModeChangeDelayRequiresAdmin() (gas: 16649)
[PASS] testSetModeChangeDelayToZero() (gas: 24610)
[PASS] testSetServiceRegistry() (gas: 31472)
[PASS] testSetServiceRegistryEmitsEvent() (gas: 30146)
[PASS] testSetServiceRegistryMultipleTimes() (gas: 644047)
[PASS] testSetServiceRegistryRejectsZeroAddress() (gas: 17294)
[PASS] testSetServiceRegistryRequiresAdmin() (gas: 19952)
[PASS] testSetSettlementToken() (gas: 536885)
[PASS] testSetSettlementTokenEmitsEvent() (gas: 539358)
[PASS] testSetSettlementTokenMultipleTimes() (gas: 1054854)
[PASS] testSetSettlementTokenRejectsZeroAddress() (gas: 16720)
[PASS] testSetSettlementTokenRequiresAdmin() (gas: 529915)
[PASS] testSetTokenWhitelistRegistry() (gas: 30878)
[PASS] testSetTokenWhitelistRegistryEmitsEvent() (gas: 29072)
[PASS] testSetTokenWhitelistRegistryMultipleTimes() (gas: 624805)
[PASS] testSetTokenWhitelistRegistryRejectsZeroAddress() (gas: 16632)
[PASS] testSetTokenWhitelistRegistryRequiresAdmin() (gas: 19072)
[PASS] testSetWithdrawalDelay() (gas: 27218)
[PASS] testSetWithdrawalDelayEmitsEvent() (gas: 24310)
[PASS] testSetWithdrawalDelayMultipleTimes() (gas: 30799)
[PASS] testSetWithdrawalDelayRequiresAdmin() (gas: 17023)
[PASS] testSetWithdrawalDelayToZero() (gas: 20465)
[PASS] testSetWithdrawalFee() (gas: 42108)
[PASS] testSetWithdrawalFeeEmitsEvent() (gas: 43809)
[PASS] testSetWithdrawalFeeRejectsExcessive() (gas: 17710)
[PASS] testSetWithdrawalFeeRequiresAdmin() (gas: 18119)
[PASS] testSetWithdrawalFeeToMax() (gas: 41272)
[PASS] testSetWithdrawalFeeToZero() (gas: 32217)
[PASS] testWhitelistManagementWorkflow() (gas: 14256568)
Suite result: ok. 45 passed; 0 failed; 0 skipped; finished in 8.23ms (7.99ms CPU time)

Ran 63 tests for test/unit/MorphoService.t.sol:MorphoServiceTest
[PASS] testBuildAuthorization() (gas: 15223)
[PASS] testBuildAuthorizationCallData() (gas: 12170)
```

```
[PASS] testBuildAuthorizationCallTarget() (gas: 13629)
[PASS] testBuildAuthorizationIgnoresAssetAndAmount() (gas: 19463)
[PASS] testBuildAuthorizationRequiresMorphoType() (gas: 14296)
[PASS] testBuildBorrow() (gas: 173304)
[PASS] testBuildBorrowCallData() (gas: 171855)
[PASS] testBuildBorrowCallTarget() (gas: 174706)
[PASS] testBuildBorrowLookupsLoanMarket() (gas: 172754)
[PASS] testBuildBorrowNoMarketReverts() (gas: 17562)
[PASS] testBuildBorrowOnBehalfAndReceiver() (gas: 177161)
[PASS] testBuildBorrowReturnsOneCall() (gas: 172380)
[PASS] testBuildBorrowSharesZero() (gas: 174178)
[PASS] testBuildDeposit() (gas: 175712)
[PASS] testBuildDepositCallData() (gas: 174770)
[PASS] testBuildDepositCallTarget() (gas: 177035)
[PASS] testBuildDepositLookupsCollateralMarket() (gas: 174045)
[PASS] testBuildDepositNoMarketReverts() (gas: 20275)
[PASS] testBuildDepositOnBehalf() (gas: 179482)
[PASS] testBuildDepositReturnsOneCall() (gas: 175509)
[PASS] testBuildRepay() (gas: 174809)
[PASS] testBuildRepayCallData() (gas: 176177)
[PASS] testBuildRepayCallTarget() (gas: 178087)
[PASS] testBuildRepayLookupsLoanMarket() (gas: 174303)
[PASS] testBuildRepayNoMarketReverts() (gas: 19059)
[PASS] testBuildRepayOnBehalf() (gas: 180630)
[PASS] testBuildRepayReturnsOneCall() (gas: 175623)
[PASS] testBuildRepaySharesZero() (gas: 177405)
[PASS] testBuildRevokeAuthorization() (gas: 13500)
[PASS] testBuildRevokeAuthorizationRequiresMorphoType() (gas: 14635)
[PASS] testBuildWithdraw() (gas: 171445)
[PASS] testBuildWithdrawCallData() (gas: 172364)
[PASS] testBuildWithdrawCallTarget() (gas: 175185)
[PASS] testBuildWithdrawLookupsCollateralMarket() (gas: 172141)
[PASS] testBuildWithdrawNoMarketReverts() (gas: 17908)
[PASS] testBuildWithdrawOnBehalfAndReceiver() (gas: 176328)
[PASS] testBuildWithdrawReturnsOneCall() (gas: 172471)
[PASS] testCollateralMarketMapping() (gas: 168806)
[PASS] testDeployment() (gas: 11558)
[PASS] testDeploymentZeroMorpho() (gas: 40061)
[PASS] testDeploymentZeroOwner() (gas: 39731)
[PASS] testGetBalance() (gas: 247273)
[PASS] testGetBalanceNoMarket() (gas: 17052)
[PASS] testGetBalanceNoPosition() (gas: 182338)
[PASS] testGetBalanceReturnsCollateralAmount() (gas: 247757)
[PASS] testGetMarketByCollateralNotRegistered() (gas: 20583)
[PASS] testGetMarketByLoanTokenNotRegistered() (gas: 18002)
[PASS] testIsMarketRegistered() (gas: 168392)
[PASS] testLoanMarketMapping() (gas: 169114)
[PASS] testMorphoSet() (gas: 9866)
[PASS] testOwnerSet() (gas: 9173)
[PASS] testProtocolId() (gas: 7217)
[PASS] testProtocolIdMatchesConstant() (gas: 6161)
[PASS] testRegisterMarket() (gas: 172114)
[PASS] testRegisterMarketAlreadyRegistered() (gas: 167279)
[PASS] testRegisterMarketComputesCorrectId() (gas: 166376)
[PASS] testRegisterMarketCreatesCollateralMapping() (gas: 168219)
[PASS] testRegisterMarketCreatesLoanMapping() (gas: 168467)
[PASS] testRegisterMarketEmitsEvent() (gas: 171772)
[PASS] testRegisterMarketOnlyOwner() (gas: 22545)
[PASS] testRegisterMarketStoresParams() (gas: 168777)
[PASS] testRegisterMarketZeroCollateral() (gas: 17776)
[PASS] testRegisterMarketZeroLoanToken() (gas: 18398)
Suite result: ok. 63 passed; 0 failed; 0 skipped; finished in 8.09ms (4.53ms CPU time)

Ran 13 tests for test/integration/ManagementAccountMorphoIntegration.t.sol:ManagementAccountMorphoIntegrationTest
[PASS] testApproveAndActivateMorphoService() (gas: 139576)
[PASS] testBorrowThroughSmartAccount() (gas: 578835)
[PASS] testCompleteWorkflow() (gas: 886380)
[PASS] testCreditModeWithMorpho() (gas: 631473)
[PASS] testDepositCollateralThroughSmartAccount() (gas: 272224)
[PASS] testMultipleDeposits() (gas: 322977)
[PASS] testMultipleSupplies() (gas: 375412)
[PASS] testRepayThroughSmartAccount() (gas: 638420)
[PASS] testSetupComplete() (gas: 68366)
[PASS] testSupplyAndWithdrawLoanToken() (gas: 362685)
[PASS] testSupplyLoanTokenToEarnYield() (gas: 319536)
```

```
[PASS] testWithdrawCollateralThroughSmartAccount() (gas: 345473)
[PASS] testWithdrawalRequestFlow() (gas: 383165)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 1.75s (5.78s CPU time)

Ran 16 tests for test/integration/MorphoServiceIntegration.t.sol:MorphoServiceIntegrationTest
[PASS] testAccount() (gas: 2858)
[PASS] testForkBuildAuthorizationCreatesValidCall() (gas: 18415)
[PASS] testForkBuildBorrowCreatesValidCall() (gas: 34944)
[PASS] testForkBuildDepositCreatesValidCall() (gas: 38329)
[PASS] testForkBuildRepayCreatesValidCall() (gas: 38776)
[PASS] testForkBuildWithdrawCreatesValidCall() (gas: 34309)
[PASS] testForkCompleteWorkflow() (gas: 494342)
[PASS] testForkExecuteAuthorizationCall() (gas: 50054)
[PASS] testForkExecuteBorrowCall() (gas: 418036)
[PASS] testForkExecuteDepositCall() (gas: 103773)
[PASS] testForkExecuteRepayCall() (gas: 459092)
[PASS] testForkExecuteRevokeAuthorizationCall() (gas: 43727)
[PASS] testForkExecuteWithdrawCall() (gas: 157873)
[PASS] testForkFullRepayment() (gas: 458732)
[PASS] testForkGetBalanceReturnsRealData() (gas: 105438)
[PASS] testForkMultipleDeposits() (gas: 131158)
Suite result: ok. 16 passed; 0 failed; 0 skipped; finished in 1.84s (5.20s CPU time)

Ran 16 test suites in 1.85s (3.69s CPU time): 418 tests passed, 0 failed, 0 skipped (418 total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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