Zellic

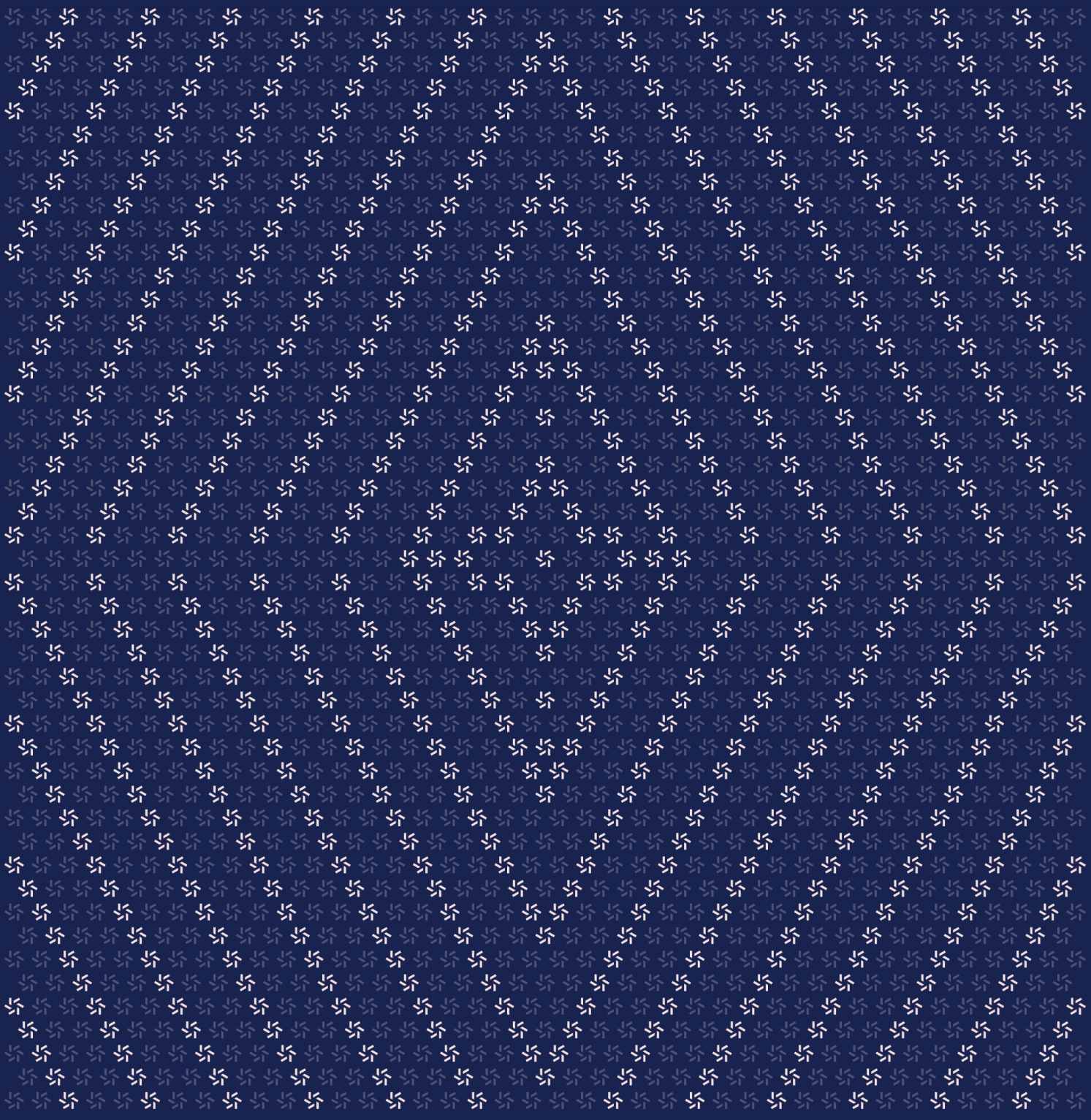
Prepared for
Othman
Hyperbeat

Prepared by
Filipe Alves
Weipeng Lai
Zellic

November 6, 2025

Hyperbeat Pay

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Hyperbeat from October 24th to October 31st, 2025. During this engagement, Zellic reviewed Hyperbeat Pay's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any vulnerabilities that could result in the loss of user funds?
- Are access controls implemented effectively to prevent unauthorized operations?
- Can users bypass restrictions and withdraw funds before their card payments settle?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Backend logic for card payment settlement
- Centralization risks, including the correctness and suitability of operator operations

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

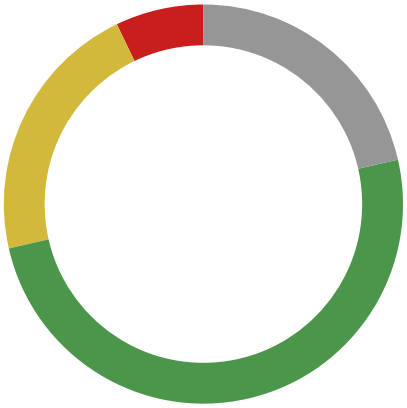
1.4. Results

During our assessment on the scoped Hyperbeat Pay contracts, we discovered 14 findings. One critical issue was found. Three were of medium impact, seven were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Hyperbeat in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	3
Low	7
Informational	1
Indeterminate	2



2. Introduction

2.1. About Hyperbeat Pay

Hyperbeat contributed the following description of Hyperbeat Pay:

> Hyperbeat Pay is a non-custodial smart account system that enables users to make card payments and settlements while maintaining full ownership of their assets, with support for both direct spending (`SPENDING` mode) and collateral-backed credit (`CREDIT` mode). The system integrates with DeFi lending protocols like Morpho Blue, allowing users to deposit collateral and authorize an operator to borrow settlement tokens on-demand for instant payment processing. Withdrawals are protected through time-locked cooldowns and operator approvals for critical tokens (settlement token in `SPENDING` mode, collateral in `CREDIT` mode), while other assets can be withdrawn instantly.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability

weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Hyperbeat Pay Contracts

Type	Solidity
Platform	EVM-compatible
Target	Liquid-Bank
Repository	https://github.com/0xhyperbeat/Liquid-Bank ↗
Version	`431de44bc37ea1b647da376bee1a3031afffff435`

Programs	`core/ManagementAccount.sol`
	`core/ManagementAccountFactory.sol`
	`core/ManagementAccountStorage.sol`
	`libraries/services/ServiceRegistryErrors.sol`
	`libraries/services/TokenWhitelistRegistryErrors.sol`
	`libraries/ManagementAccountErrors.sol`
	`libraries/ManagementAccountLib.sol`
	`libraries/ServiceErrors.sol`
	`services/morpho/MorphoService.sol`
	`services/registry/ServiceRegistry.sol`
	`services/registry/TokenWhitelistRegistry.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.7 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Filipe Alves
Engineer
filipe@zellic.io

Weipeng Lai
Engineer
weipeng.lai@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 24, 2025	Kick-off call
October 24, 2025	Start of primary review period
October 31, 2025	End of primary review period

3. Detailed Findings

3.1. The owner can self-assign the operator role

Target	ManagementAccount		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

There is an inconsistency in how ManagementAccount resolves the operator address. The contract has an `operator()` function that returns the operator from the factory, but the `_checkOperator()` and `_checkOwnerOrOperator()` functions use role-based access control instead:

```
function operator() external view returns (address) {
    return IManagementAccountFactory(_factory).operator();
}

function _checkOperator() internal view {
    if (!hasRole(OPERATOR_ROLE, msg.sender)) revert ManagementAccountErrors.
        NotOperator(msg.sender);
}

function _checkOwnerOrOperator() internal view {
    if (!hasRole(OWNER_ROLE, msg.sender) && !hasRole(OPERATOR_ROLE, msg.
        sender)) {
        revert ManagementAccountErrors.NotOperator(msg.sender);
    }
}
```

The operator is designed to have privileged access to critical functions including `settle()`, `executeModeChange()`, `approveWithdrawal()`, `approveServiceAction()`, and `approveAuthorizationRevocation()`. However, during initialization, the ManagementAccount owner is granted `DEFAULT_ADMIN_ROLE`, which allows them to freely control role assignments. This enables the owner to self-assign `OPERATOR_ROLE` and execute these critical operator-only functions or revoke the `OPERATOR_ROLE` from the legitimate factory operator entirely.

Impact

Users can approve their own withdrawal requests for critical tokens, bypass mode-change controls, and approve service actions without authorization. This fundamentally breaks the trust

model where only the legitimate operator should execute such operations.

Additionally, if the factory updates its operator address, existing ManagementAccount instances will remain unsynchronized and continue using the stale operator assigned during initialization. The new factory operator cannot act on previously deployed accounts since they will not have the `OPERATOR_ROLE` assigned, requiring manual intervention from the admin to update the operator role.

Recommendations

Remove `OPERATOR_ROLE` assignments from `initialize()` and change `_checkOperator()` and `_checkOwnerOrOperator()` to check directly against the factory's operator address:

```
function _checkOperator() internal view {
    if (!hasRole(OPERATOR_ROLE, msg.sender)) revert ManagementAccountErrors.
        NotOperator(msg.sender);
    if (msg.sender != operator()) revert ManagementAccountErrors.NotOperator(
        msg.sender);
}
[...]
function _checkOwnerOrOperator() internal view {
    if (!hasRole(OWNER_ROLE, msg.sender) && !hasRole(OPERATOR_ROLE, msg.
        sender)) {
    if (!hasRole(OWNER_ROLE, msg.sender) && msg.sender != operator()) {
          revert ManagementAccountErrors.NotOperator(msg.sender);
      }
}
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `efe2310e` ↗.

3.2. Operator change prevents revocation of prior authorizations

Target	ManagementAccount		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When the factory updates its operator address, account owners lose the ability to revoke authorizations previously granted to the old operator. Both `revokeOperatorBorrowing` and `approveAuthorizationRevocation` fetch the current operator address from the factory when constructing revocation calls:

```
function revokeOperatorBorrowing(address service) external onlyOwner {
    [...]
    address operatorAddress = IManagementAccountFactory(_factory).operator();
    IService.Call[] memory calls = IService(service).buildRevokeAuthorization(
        _creditAuthorizationType, operatorAddress, settlementTokenAddress
    );
    [...]
}

function approveAuthorizationRevocation() external onlyOperator nonReentrant {
    [...]
    address operatorAddress = IManagementAccountFactory(_factory).operator();
    IService.Call[] memory calls = IService(service).buildRevokeAuthorization(
        _creditAuthorizationType, operatorAddress, settlementTokenAddress
    );
    [...]
}
```

As a result, these functions attempt to revoke permissions for the new operator (who was never authorized) instead of the previously authorized operator, leaving the old operator's permissions intact on lending protocols.

Impact

Authorizations granted to previous operators cannot be revoked and remain active on lending protocols indefinitely. The old operator retains the ability to borrow against the account's collateral on any service where they were previously authorized.

Revocation attempts execute without reverting, providing no indication to users that the wrong operator was targeted or that the revocation failed.

Recommendations

Store the authorized operator address when authorization is granted, and then use that stored address during revocation instead of fetching from the factory:

```
mapping(address => address) private _authorizedOperators;

function authorizeOperatorBorrowing(
    address service,
    AuthorizationType authType,
    uint256 allowance
) external onlyOwner nonReentrant {
    [...]
    address operatorAddress = IManagementAccountFactory(_factory).operator();
    _authorizedOperators[service] = operatorAddress;
    [...]
}

function revokeOperatorBorrowing(address service) external onlyOwner {
    [...]
    address operatorAddress = IManagementAccountFactory(_factory).operator();
    address operatorAddress = _authorizedOperators[service];
    if (operatorAddress == address(0)) {
        revert ManagementAccountErrors.NoAuthorizationFound(service);
    }
    IService.Call[] memory calls = IService(service).buildRevokeAuthorization(
        _creditAuthorizationType, operatorAddress, settlementTokenAddress
    );
    [...]
    delete _authorizedOperators[service];
}

function approveAuthorizationRevocation() external onlyOperator nonReentrant {
    [...]
    address service = _pendingAuthRevocationService;
    address operatorAddress = IManagementAccountFactory(_factory).operator();
    address operatorAddress = _authorizedOperators[service];
    if (operatorAddress == address(0)) {
        revert ManagementAccountErrors.NoAuthorizationFound(service);
    }
```

```
    [...]
    delete _authorizedOperators[service];
}
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit 582d51b5 ↗.

3.3. Service removal prevents revocation of prior authorizations

Target	ManagementAccount		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When a service is removed from the ServiceRegistry, account owners lose the ability to revoke authorizations previously granted to that service. The `revokeOperatorBorrowing` function validates that the service is still registered, causing revocation to fail for removed services:

```
function revokeOperatorBorrowing(address service) external onlyOwner {
    _validateServiceInRegistry(service);
    [...]
}
[...]
function _validateServiceInRegistry(address service) internal view {
    IServiceRegistry registry = _registry();
    if (!registry.isServiceRegistered(service)) {
        revert ManagementAccountErrors.ServiceNotRegistered(service);
    }
    if (!registry.isServiceActive(service)) {
        revert ManagementAccountErrors.ServiceInactive(service);
    }
}
```

Impact

Authorizations granted to removed services cannot be revoked and remain active on lending protocols indefinitely. Removed or compromised services retain the ability to borrow against user collateral, with no mechanism for users to clean up these permissions.

Recommendations

Remove the registry validation check from `revokeOperatorBorrowing`:

```
function revokeOperatorBorrowing(address service) external onlyOwner {
```

```
    _validateServiceInRegistry(service);
    if (!_approvedServices.contains(service)) {
        revert ManagementAccountErrors.ServiceNotApproved(service);
    }
    [...]
}
```

The check against `_approvedServices` is sufficient to ensure the service was previously authorized.

Remediation

This issue has been acknowledged by Hyperbeat, and fixes were implemented in the following commits:

- e6ee645d ↗
- 12f34557 ↗

3.4. Credit mode changes bypass the timelock mechanism

Target	ManagementAccount		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

As shown below, `ManagementAccount::requestCreditMode` directly updates `_creditService` and `_creditCollateralToken` instead of their pending counterparts:

```
function requestCreditMode(address service, address collateralToken)
    external onlyOwner {
    [...]
   _creditService = service;
   _creditCollateralToken = collateralToken;
    [...]
}
```

This bypasses the timelock mechanism. The intended flow requires mode changes to be queued in pending state first then applied only after the cooldown period when `executeModeChange()` is called.

Impact

Users can change credit modes instantly without the required cooldown period, bypassing the timelock mechanism entirely. This eliminates the security window intended to allow operators to review mode changes and take protective actions before they become active.

Recommendations

Update `requestCreditMode` to set pending state variables:

```
function requestCreditMode(address service, address collateralToken)
    external onlyOwner {
     [...]
    _creditService = service;
    _creditCollateralToken = collateralToken;
```

```
    _pendingCreditService = service;
    _pendingCreditCollateralToken = collateralToken;
      [...]
}
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit 73c583a8 ↗.

3.5. Incorrect selector extraction always returns zero

Target	ManagementAccountLib		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `extractSelector` function in ManagementAccountLib incorrectly applies a `shr(224, ...)` operation before assigning to a `bytes4` return type:

```
function extractSelector(bytes memory data)
    internal pure returns (bytes4 selector) {
    if (data.length < 4) return bytes4(0);
    assembly {
        selector := shr(224, mload(add(data, 32)))
    }
}
```

The `shr(224, ...)` shifts the selector bytes to the right, moving them out of the `bytes4` range entirely, resulting in `bytes4(0)`. The shift is unnecessary since casting to `bytes4` already extracts the first four bytes from the loaded `bytes32` word.

This function is used in `executeApprovedServiceAction` and `_executeServiceCalls` to emit the `ServiceCallExecuted` event:

```
function _executeServiceCalls(
    address service,
    IService.Call[] memory calls
)
    internal
    returns (bytes[] memory results)
{
    [...]
    // Emit events for each call
    uint256 length = calls.length;
    for (uint256 i; i < length; ++i) {
        bytes4 selector
    = ManagementAccountLib.extractSelector(calls[i].callData);
        emit ServiceCallExecuted(service, calls[i].target, selector);
    }
```

```
}

function executeApprovedServiceAction(
    uint256 actionId,
    string calldata action
)
    external
    nonReentrant
    returns (bytes[] memory results)
{
    [...]
    for (uint256 i; i < length; ++i) {
        IService.Call memory call = calls[i];
        [...]
        bytes4 selector = ManagementAccountLib.extractSelector(call.callData);
        emit ServiceCallExecuted(service, call.target, selector);

        results[i] = response;
    }
    [...]
}
```

Due to this bug, the event always emits `bytes4(0)` instead of the actual function selector.

Impact

This breaks off-chain monitoring that relies on these events to track which functions are executed during approved service actions.

Recommendations

Remove the unnecessary `shr(224, ...)` operation from the assembly block:

```
function extractSelector(bytes memory data)
    internal pure returns (bytes4 selector) {
    if (data.length < 4) return bytes4(0);
    assembly {
        selector := shr(224, mload(add(data, 32)))
        selector := mload(add(data, 32))
    }
}
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `d7b521b0`.

3.6. Spending limit can exceed available settlement balance

Target	ManagementAccount		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `increaseSettlementAllowance` function allows owners to set spending limits that exceed their available settlement-token balance in `SPENDING` mode:

```
function increaseSettlementAllowance(uint256 additionalAmount)
    external override onlyOwner {
    if (additionalAmount == 0)
    revert ManagementAccountErrors.InvalidAmount(additionalAmount);

    _spendingLimit.settlementAllowance += additionalAmount;

    emit SettlementAllowanceIncreased(additionalAmount,
    _spendingLimit.settlementAllowance);
}
```

This creates an inconsistency with `executeModeChange`, which enforces that available balance must be greater than or equal to the spending limit when switching to `SPENDING` mode:

```
function executeModeChange() external onlyOperator {
    [...]
    // Enforce settlement token minimum when switching TO SPENDING mode
    if (_pendingMode == AccountMode.SPENDING &&
    _spendingLimit.settlementAllowance > 0) {
        address settlementTokenAddress = _settlementToken();
        uint256 availableBalance = _availableBalance(settlementTokenAddress);
       if (availableBalance < _spendingLimit.settlementAllowance) {
            revert
    ManagementAccountErrors.SettlementBalanceBelowSpendingLimit(
                availableBalance, _spendingLimit.settlementAllowance
            );
        }
    }
    [...]
```

Impact

Users in `SPENDING` mode can set spending limits that exceed their available settlement balance, violating the protocol's invariant that balance is greater than or equal to the spending limit in `SPENDING` mode. This creates several issues.

Off-chain payment processing systems relying on the spending limit to authorize card payments may approve transactions that fail during settlement, leading to payment failures and poor user experience.

Additionally, this inconsistency creates a confusing state where users can bypass balance checks by increasing their allowance while in `SPENDING` mode, even though they would be blocked from entering `SPENDING` mode with the same allowance/balance ratio.

Recommendations

Add validation to ensure the new spending limit does not exceed the available settlement balance in `SPENDING` mode:

```
function increaseSettlementAllowance(uint256 additionalAmount)
    external override onlyOwner {
     if (additionalAmount == 0)
    revert ManagementAccountErrors.InvalidAmount(additionalAmount);


    uint256 newAllowance = _spendingLimit.settlementAllowance + additionalAmou
        nt;
    address settlementTokenAddress = _settlementToken();
    uint256 availableBalance = _availableBalance(settlementTokenAddress);

    if (_mode == AccountMode.SPENDING && newAllowance > availableBalance) {
        revert ManagementAccountErrors.SettlementAllowanceExceedsBalance(
            newAllowance, availableBalance);
    }

     _spendingLimit.settlementAllowance += additionalAmount;
     _spendingLimit.settlementAllowance = newAllowance;

     emit SettlementAllowanceIncreased(additionalAmount,
    _spendingLimit.settlementAllowance);
 }
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `8afead3e`.

3.7. Balance calculation in MorphoService does not account for loan market assets

Target	MorphoService		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `getBalance` function in MorphoService only accounts for collateral positions and ignores assets supplied to loan markets for earning interest:

```
// ManagementAccount.sol
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;
        try IService(service).getBalance(token, address(this))
    returns (uint256 serviceBalance) {
            balance += serviceBalance;
        } catch { }
    }
}

// MorphoService.sol
function getBalance(address asset, address account)
    external view override returns (uint256 balance) {
   Id marketId = collateralMarket[asset];
    if (!isMarketRegistered[marketId]) return 0;

    Position memory pos = morpho.position(marketId, account);
   balance = pos.collateral;
}
```

When users supply assets to Morpho loan markets to earn interest, those positions are not included in the balance calculation returned by `MorphoService::getBalance`. This causes

`ManagementAccount::getBalance` to underreport the account's actual balance when aggregating across services.

Impact

Users see incorrect balance information in UI and off-chain systems that rely on `getBalance` for display purposes. The actual on-chain functionality remains unaffected as critical operations use direct balance queries.

Recommendations

Update `MorphoService::getBalance` to include both collateral and supply positions. Note that supply positions in Morpho are denominated in shares, which should be converted to asset amounts:

```
function getBalance(address asset, address account)
   external view override returns (uint256 balance) {
    Id marketId = collateralMarket[asset];
   if (!isMarketRegistered[marketId]) return 0;

   Position memory pos = morpho.position(marketId, account);
   balance = pos.collateral;
   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]) {
       MarketParams memory params = marketParamsById[loanMarketId];
       balance += MorphoBalancesLib.expectedSupplyAssets(morpho, params,
           account);
   }
}
```

Note that this change could break the existing usage of `getBalance` in `ManagementAccountLib::executeServiceCalls`.

To address this, we recommend introducing a dedicated `getCollateralBalance` function that returns only the collateral market token balance. Then, update `ManagementAccountLib::executeServiceCalls` to use `getCollateralBalance` in place of

`getBalance` wherever the collateral market balance is specifically required.

Remediation

This issue has been acknowledged by Hyperbeat, and fixes were implemented in the following commits:

- `2a47ed7d` ↗
- `dee56b0b` ↗

3.8. Incorrect debt and health-factor calculations in MorphoService

Target	MorphoService		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

Both `getDebt` and `getHealthFactor` functions contain calculation errors that underestimate debt and produce incorrect health factors.

Debt calculation. The debt calculation uses `totalBorrowAssets`, which reflects only the last cached state without pending accrued interest and rounds down instead of up (favoring the borrower):

```
function getDebt(address account)
    external view override returns (uint256 debt) {
    [...]
    debt = uint256(pos.borrowShares) * uint256(m.totalBorrowAssets)
    / uint256(m.totalBorrowShares);
}
```

This issue affects both `getDebt` and `getHealthFactor` functions.

Health-factor collateral calculation. The `getHealthFactor` function incorrectly calculates collateral value by using raw collateral amount instead of converting it to loan-token terms using oracle prices:

```
function getHealthFactor(address account)
    external view override returns (uint256) {
    [...]
    // collateralValue = collateral * lltv (loan-to-value)
    // Note: In production, you'd need to get oracle price and compute properly
    // This is simplified for the example
    uint256 maxBorrow = uint256(pos.collateral) * params.lltv / 1e18;
    [...]
}
```

The comment acknowledges this limitation, but the function is used in production without proper oracle price conversion.

Impact

Users and off-chain systems see incorrect debt and health-factor information. This may lead users to believe they owe less than they actually do or have inaccurate risk assessments of their positions. It primarily affects UI displays and monitoring systems.

Recommendations

To address the calculation errors, use `MorphoBalancesLib::expectedBorrowAssets` for debt calculations in both functions, (as recommended in Morpho's documentation ↗, which properly accounts for accrued interest and implements correct rounding behavior), and implement proper oracle price conversion for collateral value in `getHealthFactor`.

Alternatively, consider removing `getDebt` and `getHealthFactor` if they are only used for testing, to avoid confusion from inaccurate calculations.

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `850808eb` ↗. The `getDebt` and `getHealthFactor` functions have now been removed.

3.9. ManagementAccount returns a stale owner after ownership transfer

Target	ManagementAccount		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

ManagementAccount uses OpenZeppelin's AccessControl for permissions. Functions protected by the `onlyOwner` modifier check `OWNER_ROLE` via `hasRole()`:

```
function _checkOwner() internal view {
    if (!hasRole(OWNER_ROLE, msg.sender)) revert ManagementAccountErrors.
        NotOwner(msg.sender);
}
```

However, the `owner()` view function returns a cached `_owner` storage variable that is only set during initialization and never updated:

```
function initialize(address owner_, address factory_) external initializer {
    [...]
    _owner = owner_;
    [...]
    _setupRole(DEFAULT_ADMIN_ROLE, owner_);
    _setupRole(OWNER_ROLE, owner_);
    [...]
}

function owner() external view returns (address) {
    return _owner;
}
```

Since `DEFAULT_ADMIN_ROLE` can grant `OWNER_ROLE` to new addresses and revoke it from the original owner, the actual owner can change through role management. When this happens, `_checkOwner()` enforces the new role holder, but `owner()` continues returning the original address, creating a mismatch between the view function and actual access control.

Impact

Any off-chain or on-chain systems relying on the owner to identify the current account owner will receive incorrect information, potentially leading to authorization failures, incorrect UI displays, or operational issues.

Recommendations

Update `owner()` to derive the owner from `OWNER_ROLE` instead of the storage variable:

```
function owner() external view returns (address) {
    return _owner;
    // Get the first member of OWNER_ROLE

      // Note: This assumes single owner; adjust if multiple owners are supported
    return getRoleMember(OWNER_ROLE, 0);
}
```

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `811642af` ↗. The `owner` view function has now been removed.

3.10. Morpho authorization ignores the `amount` parameter

Target	MorphoService		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

As shown below, `ManagementAccount::authorizeOperatorBorrowing` accepts an `amount` parameter intended to limit the operator's borrowing capacity:

```
function authorizeOperatorBorrowing(
    address service,
    IService.AuthorizationType authType,
    uint256 amount
)
    external
    override
    onlyOwner
    nonReentrant
    returns (bytes[] memory results)
{
    [...]
    IService.Call[] memory calls =
        IService(service).buildAuthorization(authType, operatorAddress,
            settlementTokenAddress, amount);
    results = _executeServiceCalls(service, calls);
    [...]
    emit OperatorAuthorizedForBorrowing(operatorAddress, service, authType,
    amount);
}
```

However, `MorphoService::buildAuthorization` ignores this parameter entirely and grants unlimited borrowing delegation:

```
function buildAuthorization(
    AuthorizationType authType,
    address delegate,
    address asset,
    uint256 amount
```

```
)
    external
    view
    override
    returns (Call[] memory calls)
{
    require(authType == AuthorizationType.MORPHO_AUTHORIZATION,
    "MorphoService: invalid auth type");

    calls = new Call[](1);
    calls[0] = Call({
        target: address(morpho),
        callData: abi.encodeWithSignature("setAuthorization(address,bool)",
    delegate, true),
        value: 0
    });
}
```

Morpho Blue's authorization mechanism is binary (authorized or not) and does not support amount-limited delegation. Once authorized, the operator can borrow any amount up to the position's collateral capacity. A user calling `authorizeOperatorBorrowing(service, authType, 1000e6)` expecting to limit the operator to borrowing 1,000 USDC will instead grant unlimited borrowing rights.

Impact

This creates a mismatch between user expectations and actual authorization scope. The function signature and emitted event suggest amount-based limits exist, potentially leading users to make trust decisions based on false assumptions about borrowing restrictions. While the operator is a trusted party, users lack visibility into the actual unlimited nature of the authorization.

Recommendations

Either remove the amount parameter from the `buildAuthorization` interface for Morpho services to eliminate confusion, or implement application-layer controls to track and enforce borrowing limits if amount restrictions are desired.

Remediation

This issue has been acknowledged by Hyperbeat.

Hyperbeat provided the following response to this finding:

We will need it for other protocols

3.11. Missing access control on `createAccount` enables unlimited proxy-deployment spam

Target	ManagementAccountFactory		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `ManagementAccountFactory::createAccount` function lacks access control, unlike other administrative functions in the factory:

```
function createAccount(address owner_) external returns (address account) {
    bytes memory initData
    = abi.encodeWithSelector(ManagementAccount.initialize.selector, owner_,
    address(this));
   account = address(new ERC1967Proxy(implementation, initData));

    emit AccountCreated(account, owner_);
}
```

This allows anyone to create unlimited accounts for arbitrary addresses without restriction.

Impact

Without access control, attackers can create unlimited accounts for arbitrary addresses, enabling spam and griefing attacks.

Spam attacks. Unlimited account creation can overwhelm backend infrastructure tracking accounts, causing database bloat, increased costs, and potential denial-of-service on indexing services.

Griefing attacks. Accounts can be created for arbitrary addresses without consent, associating unwanted Hyperbeat Pay accounts with user addresses.

Recommendations

Restrict `createAccount` to an authorized role, consistent with other factory administrative functions:

```
function createAccount(address owner_) external returns (address account) {
function createAccount(address owner_) external onlyRole(IMPLEMENTATION_
    ADMIN_ROLE) returns (address account) {
     bytes memory initData
    = abi.encodeWithSelector(ManagementAccount.initialize.selector, owner_,
    address(this));
     account = address(new ERC1967Proxy(implementation, initData));

     emit AccountCreated(account, owner_);
 }
```

This allows the backend to enforce validation, rate limiting, and any necessary authorization checks before account deployment.

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit b0a7e5e9 ↗.

3.12. Functions allow interaction with services removed from registry

Target	ManagementAccount		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Informational

Description

Several functions validate services using only the local `_approvedServices[service]` mapping without checking the ServiceRegistry, allowing interaction with previously approved services that have been removed from the registry:

```
function getBalance(address service, address asset)
    external view returns (uint256) {

    if (!_approvedServices[service]) revert ManagementAccountErrors.ServiceNotA
        pproved(service);
    [...]
}

function requestCreditMode(address service, address collateralToken)
    external onlyOwner {

    if (!_approvedServices[service]) revert ManagementAccountErrors.ServiceNotA
        pproved(service);
    [...]
}

function requestCreditModeChange(address newService,
    address newCollateralToken) external onlyOwner {
   if (!_approvedServices[newService]) revert ManagementAccountErrors.
        ServiceNotApproved(newService);
    [...]
}
```

This creates inconsistent behavior where some functions (like `executeServiceAction`) properly validate registry membership using `_validateRegisteredService`, while these functions only check local approval status.

Impact

Accounts can continue interacting with delisted or compromised services through `getBalance`, `requestCreditMode`, `requestCreditModeChange`, and `executeModeChange`. This undermines the registry's purpose as a centralized control point for managing active services and creates security risks if services are removed due to vulnerabilities or deprecation.

Recommendations

Update these functions to use `_validateRegisteredService` for proper validation:

```
function getBalance(address service, address asset)
    external view returns (uint256) {
    if (!_approvedServices[service]) revert ManagementAccountErrors.
        ServiceNotApproved(service);
    _validateRegisteredService(service);
    [...]
}

function requestCreditMode(address service, address collateralToken)
    external onlyOwner {
    if (!_approvedServices[service]) revert ManagementAccountErrors.
        ServiceNotApproved(service);
    _validateRegisteredService(service);
     [...]
}

function requestCreditModeChange(address newService,
    address newCollateralToken) external onlyOwner {
    if (!_approvedServices[newService]) revert ManagementAccountErrors.
        ServiceNotApproved(newService);
    _validateRegisteredService(newService);
     [...]
}

function executeModeChange() external onlyOperator {
    if (_pendingMode == AccountMode.CREDIT && _pendingCreditService !=
        address(0)) {
        _validateRegisteredService(_pendingCreditService);
    }
     [...]
}
```

Remediation

This issue has been acknowledged by Hyperbeat.

3.13. Double-spend attack in `CREDIT` mode through collateral-withdrawal race condition

Target	ManagementAccount		
Category	Business Logic	**Severity**	Critical
Likelihood	N/A	**Impact**	Indeterminate

Description

A malicious user can execute a double-spend attack in `CREDIT` mode by withdrawing collateral between card authorization and settlement, causing the operator's borrowing attempt to fail.

This is the attack sequence:

1. A user supplies collateral into the service then requests a collateral withdrawal via `executeServiceAction(service, "withdraw", params)`.
2. Once the operator approves the withdrawal, the user makes a purchase using card payment.
3. The user immediately calls `executeApprovedServiceAction(actionId, "withdraw")` to withdraw collateral from the service to the account.
4. If step 3 executes before the operator processes the authorized borrowing, the operator will fail to borrow on behalf of the account and cannot settle the card payment.
5. The user calls `requestWithdrawal`. Since the account is in `CREDIT` mode and the token is no longer collateral in the service, the withdrawal executes instantly without operator approval.

The user completes a card payment without the operator being able to settle it, effectively receiving goods/services for free.

Impact

This enables direct financial loss where users can obtain goods or services without payment being settled. The operator cannot recover the funds since the collateral has been withdrawn before borrowing could occur. While the card payment backend could potentially prevent this through proper sequencing checks, the contract-level vulnerability allows the attack vector to exist.

The overall impact is indeterminate as it highly depends on backend implementation and operational controls. If the card payment system validates that no pending withdrawals exist before authorizing payments, this vulnerability can be effectively mitigated at the application layer.

However, if such controls are absent, this becomes a critical issue enabling systematic double-spending with direct financial losses. The contract-level design permits this attack pattern, making security entirely reliant on off-chain safeguards.

Recommendations

To prevent this attack, consider the following approaches:

1. The card payment backend should verify that there are no pending withdrawal-service actions before authorizing card payments.
2. The operator should approve and execute requested collateral withdrawal-service actions in a single call to prevent the race-condition window.

Remediation

This issue has been acknowledged by Hyperbeat.

Hyperbeat provided the following response to this finding:

> Managed on the backend side

3.14. Conflicting authorization-revocation flows

Target	ManagementAccount		
Category	Business Logic	**Severity**	Indeterminate
Likelihood	N/A	**Impact**	Indeterminate

Description

Two conflicting authorization-revocation flows exist in the system with different access-control requirements:

The **first flow** is a two-step process with operator approval.

```
// Owner initiates revocation
requestRevokeServiceAuthorization(service)
    → creates pending revocation with cooldown
    → operator must approve via approveAuthorizationRevocation()
```

The **second flow** is direct revocation without operator approval.

```
// Owner directly revokes
revokeOperatorBorrowing(service)
    → immediately revokes authorization
    → bypasses operator approval entirely
```

The existence of `revokeOperatorBorrowing` allows owners to bypass the intended two-step revocation process that requires operator oversight. This creates an inconsistency in the authorization model where one path enforces operator approval (suggesting it is a critical security control) while another path allows owners to act unilaterally.

Impact

The impact is indeterminate as it depends on the intended design:

- If operator approval is required for revocation (to ensure settlements are complete before removing authorization), then `revokeOperatorBorrowing` undermines this control.
- If owners should have unilateral revocation rights, then the two-step process in `requestRevokeServiceAuthorization` adds unnecessary complexity.

The inconsistency creates confusion about the actual authorization model and may lead to incorrect assumptions about access controls.

Recommendations

Clarify the intended authorization revocation model:

1. If operator approval is required, remove `revokeOperatorBorrowing` to enforce the two-step process consistently.
2. If owners should have direct revocation rights, remove `requestRevokeServiceAuthorization` and `approveAuthorizationRevocation`, keeping only `revokeOperatorBorrowing`.

Remediation

This issue has been acknowledged by Hyperbeat, and a fix was implemented in commit `582d51b5` ↗. Direct revocation of the operator's credit service authorization in CREDIT mode is now not allowed.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Incompatible with nonstandard ERC-20 tokens

The codebase is designed to work exclusively with standard-compliant ERC-20 tokens. Integrating tokens with nonstandard behaviors can lead to functional issues and increased risk. The following token characteristics are particularly problematic.

Missing return value. Some widely used tokens (such as USDT on Ethereum) do not return a boolean value for `transfer` or `transferFrom` calls as specified by the ERC-20 standard. Since the codebase explicitly checks for a `true` return value, these tokens will not function as intended and may cause transaction failures.

Fee on transfer. Certain tokens implement transfer fees that deduct a portion of the transacted amount. Such behavior disrupts internal balance accounting and can break core protocol invariants.

Reentrancy via callbacks. Tokens with callback hooks (e.g., ERC-777 tokens) can trigger external calls during transfers, increasing the attack surface. The current codebase is not designed to safely support such reentrant behaviors.

Given these incompatibilities, we recommend that only well-audited, standard ERC-20 tokens are whitelisted. Tokens with any of the features described above should be excluded to ensure protocol safety and correct operation.

4.2. Withdrawal-fee bypass with dust amounts

The withdrawal-fee calculation lacks a minimum withdrawal-amount check. When `request.amount * withdrawalFee < 10_000`, the fee calculation rounds down to zero, allowing fee-free withdrawals:

```
if (withdrawalFee > 0 && feeRecipient != address(0)) {
    feeAmount = (uint256(request.amount) * withdrawalFee) / 10_000;
    amountAfterFee = request.amount - feeAmount;
    [...]
}
```

For standard tokens with six or more decimals (e.g., USDC with six decimals), this requires extremely small withdrawal amounts to trigger the bypass. For example, with a 1% fee (100 basis points), amounts below 100 token units (0.0001 USDC) would incur no fee. The real-world impact is

minimal for such tokens.

However, if the protocol whitelists tokens with lower decimal precision in the future, the bypass becomes more significant. A token with two decimals and a 1% fee would allow fee-free withdrawals for any amount below one full token unit.

We recommend enforcing a minimum withdrawal amount in `requestWithdrawal` if the protocol plans to support tokens with lower decimal precision. Alternatively, document that only tokens with sufficient decimal precision should be whitelisted to make this edge case economically insignificant.

4.3. Missing token whitelist validation in `ACTION_CUSTOM` path

The `ACTION_CUSTOM` path in `ManagementAccount::executeServiceAction` does not validate whether involved tokens are whitelisted in the `TokenWhitelistRegistry`. While standard actions (deposit, withdraw, borrow, repay) explicitly check token whitelist status, custom actions bypass this validation:

```
function executeServiceAction(
    address service,
    string calldata action,
    bytes calldata params
)
    external
    override
    onlyOwner
    nonReentrant
    returns (bytes[] memory results)
{
    [...]
    if (actionHash == ACTION_DEPOSIT) {
        (address asset, uint256 amount, bool isCollateral)
    = abi.decode(params, (address, uint256, bool));
       if (!_tokenWhitelistRegistry().isTokenWhitelisted(asset)) {
            revert ManagementAccountErrors.TokenNotWhitelisted(asset);
        }
        calls = IService(service).buildDeposit(asset, amount, isCollateral);
    }
    [...]
    else if (actionHash == ACTION_CUSTOM) {
       calls = IService(service).buildCustom(params);
    }
    [...]
}
```

The responsibility for token validation is delegated to the service's `buildCustom` implementation. As of the time of writing, `MorphoService::buildCustom` returns an empty call array, and the protocol does not use custom actions. However, if future services implement `buildCustom` functionality, they should ensure proper token whitelist validation to maintain consistent security controls across all action types.

We recommend documenting in the IService interface that `buildCustom` implementations must validate token addresses against the whitelist registry if they involve token operations, ensuring consistent security guarantees across standard and custom actions.

4.4. Test suite

The test suite adequately covers the user flows in ManagementAccount. However, there are no unit tests specifically for ManagementAccountLib. The issue in Finding 3.5. ↗ could have been caught by a direct unit test.

The test coverage for this project should be expanded to include all contracts and libraries, not just surface-level functions. We recommend building a rigorous test suite that includes all contracts and libraries to ensure that the system operates securely and as intended.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: ManagementAccount

Description

The ManagementAccount is a noncustodial smart contract wallet that serves as the core of the Hyperbeat Pay system. It allows users to maintain custody of their crypto assets while enabling card payment settlements through the Hyperbeat backend. It operates in two distinct modes: `SPENDING` mode, which directly debits the settlement token held in the account for card payments, and `CREDIT` mode, which authorizes the operator to borrow settlement tokens on behalf of the user against collateral deposited in services.

The ManagementAccount is responsible for the following:

- **Asset management.** Users can deposit funds into the account via the `deposit` function or by directly transferring tokens to the contract. The available balance is calculated as the current settlement-token balance in the contract minus the amount requested for withdrawal. For withdrawals, settlement-token withdrawals in `SPENDING` mode require a cooldown period and operator approval after the cooldown expires. All other token withdrawals are instant. Withdrawal fees are deducted from the withdrawal amount when finalizing the withdrawal.
- **Settlement operations.** The operator can call `settle` to debit settlement tokens for card payments; `settle` consumes the spending-limit allowance configured by the user. In `CREDIT` mode, the user must authorize the operator to borrow settlement tokens on behalf of the user.
- **Service integration.** Services (e.g., MorphoService) implement the IService interface and return `Call[]` arrays that the ManagementAccount executes. The ManagementAccount maintains a list of approved services and executes protocol-specific sequences by requesting a service to build low-level calls (deposit/withdraw/borrow/repay/custom) that it then executes.

Invariants

Settlement protection

- Operator-settlement transfers must not exceed the remaining spending allowance.
- In `SPENDING` mode, the available settlement-token balance must remain above the spending allowance configured by the user.

- In `SPENDING` mode, user withdrawals of the settlement token require a cooldown period and operator approval before processing. The Hyperbeat backend must ensure there are no pending card payments before approving such withdrawals.
- In `CREDIT` mode, user withdrawals from the current credit service to the account require cooldown and operator approval before processing. The Hyperbeat backend must ensure there are no pending card payments before approving such withdrawals.
- Mode changes require operator approval. The Hyperbeat backend must ensure there are no pending card payments before approving mode changes.

Token whitelist enforcement

- Any token used in deposits, service actions, settlement, or withdrawals must be whitelisted by the TokenWhitelistRegistry.
- The settlement-token address must be retrieved from the factory.

Service integrity

- A service must be registered and active in the ServiceRegistry and explicitly approved by the owner before use; unapproved services cannot be called.
- The credit service cannot be revoked if it is the current `_creditService` and the account is in `CREDIT` mode.

Upgrade safety

- Upgrades are `UPGRADER_ROLE`-gated and only allowed to implementations whitelisted by the factory.

Attack surface

- **Operator approval-restriction bypass.** Users can change the operator, which could bypass the approval restrictions enforced by the `onlyOperator` modifier (see Finding 3.1. ↗).
- **The `token` input surface.** The `token` input is validated to ensure it is whitelisted in the TokenWhitelistRegistry during deposit and withdrawal operations, preventing users from inputting malicious token addresses.
- **The `service` input surface.** The `_validateRegisteredService` check in `executeServiceAction` ensures the `service` is registered and active in the ServiceRegistry, preventing users from inputting malicious services for actions.
- **Contract upgrade surface.** Upgrade implementations are limited by the factory whitelist, preventing arbitrary implementations from being set.

5.2. Component: ManagementAccountFactory

Description

The ManagementAccountFactory is an upgradable factory contract responsible for deploying ManagementAccount instances. It serves as the central configuration hub for the Hyperbeat Pay system, maintaining global parameters that all ManagementAccount instances reference. The factory deploys new accounts using a proxy pattern and enforces implementation whitelisting to ensure secure upgrades.

Invariants

Initialization safety

- The factory can only be initialized once.
- The constructor disables initializers to prevent initialization of the implementation contract.

Access control

- Only `IMPLEMENTATION_ADMIN_ROLE` can modify configuration parameters, manage the implementation whitelist, and authorize factory upgrades.
- The `DEFAULT_ADMIN_ROLE` can grant or revoke the `IMPLEMENTATION_ADMIN_ROLE`.

Implementation whitelist enforcement

- Only whitelisted implementations can be set as the active implementation via `setImplementation`.
- The currently active implementation cannot be removed from the whitelist.

Attack surface

- **Permissionless `createAccount`.** The `createAccount` function lacks access control, allowing any external party to deploy an unlimited number of account instances for a specific `owner` (see Finding 3.11. ↗).
- **Configuration updates.** Only `IMPLEMENTATION_ADMIN_ROLE` can update the configurations, preventing malicious external parties from modifying the global configurations.
- **Operator replacement.** Only `IMPLEMENTATION_ADMIN_ROLE` can update the operator address, preventing malicious external parties from modifying the operator.
- **Factory upgrade.** Only `IMPLEMENTATION_ADMIN_ROLE` can upgrade the factory, preventing malicious external parties from performing unauthorized upgrades.

5.3. Component: ServiceRegistry

Description

The ServiceRegistry is a centralized access-control registry that maintains a curated list of DeFi protocol service adapters available for ManagementAccount interactions. It establishes the system's trust boundary by ensuring that only validated services can be used by ManagementAccounts to interact with external DeFi protocols. Each registered service contains metadata including its service type (`LENDING`, `BORROWING`, `YIELD`, or `CUSTOM`), protocol identifier, risk score, and active status.

Invariants

Access control

- Only accounts with `SERVICE_ADMIN_ROLE` can register, update, or remove services.
- The `DEFAULT_ADMIN_ROLE` can grant or revoke the `SERVICE_ADMIN_ROLE`.

Attack surface

- **Service deactivation risk.** A `SERVICE_ADMIN_ROLE` can deactivate services by setting `active` to false or removing them entirely, which would prevent all ManagementAccounts from executing new actions with that service (see Finding 3.3. ↗).
- **Service management.** Only `SERVICE_ADMIN_ROLE` can manage services, preventing malicious external parties from registering, upgrading, or removing services.

5.4. Component: TokenWhitelistRegistry

Description

The TokenWhitelistRegistry is a central access-control registry that manages which tokens are permitted for use across all ManagementAccounts. It serves as the single source of truth for token whitelisting, ensuring that only approved tokens can be deposited, withdrawn, used in service actions, or utilized for settlements.

Invariants

Settlement-token enforcement

- The settlement token is immutable and automatically whitelisted upon deployment.
- The settlement token cannot be removed from the whitelist.

Access control

- Only addresses with the `TOKEN_ADMIN_ROLE` can whitelist new tokens or remove existing tokens.
- The `DEFAULT_ADMIN_ROLE` can grant or revoke the `TOKEN_ADMIN_ROLE`.

Attack surface

- **Token removal risk.** A `TOKEN_ADMIN_ROLE` can remove a token from the whitelist, which would prevent all ManagementAccounts from withdrawing that token from their accounts. This risk is mitigated by the fact that the settlement token cannot be removed from the whitelist.
- **Token whitelist management.** Only `TOKEN_ADMIN_ROLE` can manage the token whitelist, preventing malicious external parties from adding or removing tokens from the whitelist.

5.5. Component: MorphoService

Description

The MorphoService is a service adapter contract that enables `ManagementAccounts` to interact with the Morpho lending protocol. It constructs Morpho-specific call payloads that `ManagementAccounts` can execute.

Invariants

Market registration integrity

- Only the service owner can register markets.
- Once registered, market parameters stored in `marketParamsById` cannot be modified.

Asset verification

- When building action calldata for a collateral asset, the function verifies that the market for that collateral asset is already registered.
- When building action calldata for a loan asset, the function verifies that the market for that loan asset is already registered.

Authorization integrity

- Only the `MORPHO_AUTHORIZATION` type is supported for authorization operations.
- Morpho authorization for the operator is binary (granted or revoked) and does not support granular amounts.

Attack surface

- **The `asset` input validation.** The `asset` parameter in build functions is validated by

verifying it maps to a registered market. Unregistered assets cause transactions to revert, preventing interaction with unsupported tokens.

6. Assessment Results

During our assessment on the scoped Hyperbeat Pay contracts, we discovered 14 findings. One critical issue was found. Three were of medium impact, seven were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.