



Security Review For Blend Money



Collaborative Audit Prepared For: **Blend Money**
Lead Security Expert(s): [eyore](#)
[montecristo](#)
Date Audited: **January 12 - January 15, 2026**

Introduction

Blend is an intent-driven, non-custodial Yield Coordination Engine. It functions as a savings account for your digital assets: you keep full custody in a personal Gnosis Safe while Blend's automation allocates capital between stable base yields and sophisticated, delta-neutral strategies.

This architecture provides the simplicity of a “set-and-forget” yield app. Blend achieves this by building on top of trusted DeFi primitives, not by replacing them.

Scope

Repository: BlendMoney/contracts

Audited Commit: f49561315c69543d3b85144111616eadbb159301

Final Commit: 072911df5e95252e8c98c0c4ef5041bcccd41359

Files:

- src/adapters/OstiumVaultController.sol
- src/adapters/SwapAdapter.sol
- src/adapters/VaultToVaultAction.sol
- src/bundler3-adapters/BalanceReplacementAdapter.sol
- src/libraries/BasePerpVaultController.sol
- src/libraries/MorphoVaultLib.sol
- src/libraries/PriceLib.sol
- src/libraries/VaultController.sol
- src/types/RebalanceTypes.sol
- src/types/StrategyTypes.sol

Final Commit Hash

072911df5e95252e8c98c0c4ef5041bcccd41359

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

| High | Medium | Low/Info |
|------|--------|----------|
| 0 | 2 | 17 |

Issues Not Fixed and Not Acknowledged

| High | Medium | Low/Info |
|------|--------|----------|
| 0 | 0 | 0 |

Issue M-1: Flaw in totalAvailable validation during executeRebalance [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/11>

Summary

The `totalAvailable` check in `BasePerpsVaultController::executeRebalance` contains logic flaws that cause unnecessary transaction reverts. For the Ostium platform, it fails to account for upfront oracle fees when closing positions to increase them, while for other platforms, it ignores capital that will be freed during the same operation's internal position adjustments.

Vulnerability Detail

`BasePerpsVaultController::executeRebalance` execution flow can be summarized as the following:

1. Consolidate Safe collateral into vault
2. Process closes (`isIncrease = false`)
3. Consolidate freed capital from closes
4. Validate sum of opens \leq available balance
5. Process opens (`isIncrease = true, amount > 0`)
 1. Close existing position first (if any)
 2. Open new position with specified collateral amount

There are two problems with available balance validation (4)

Ostium Platform: Missing Oracle Fee Consideration

On Ostium, closing a position requires paying an oracle fee in collateral upfront (later refunded by an keeper). When step 5.i executes to close an existing position before increasing it, this oracle fee must be available.

At the time of validation (step 4), vault's total asset equals to the following:

```
totalAsset = previousTotalAsset + freedCapitalFromCloses -  
            oracleFeeForDecreasingPositionsClose
```

, where `freedCapitalFromCloses = 0` (refund occurs later).

The actual collateral requirement for step 5 is:

```
totalRequired = oracleFeeForIncreasingPositionsClose +  
    ↳ totalCollateralForNewPositions
```

However, 4 only checks:

```
totalAsset >= totalCollateralForNewPositions
```

completely ignoring the `oracleFeeForIncreasingPositionsClose`.

Other Platforms: Ignoring Internally Freed Capital

For platforms where closing provides instant refunds, validation at step 4 fails to account for capital that will be freed during step 5.i when closing positions to increase them.

Consider this example scenario:

- Position A: 300 collateral
- Position B: 400 collateral
- Vault balance: 100 collateral
- Desired rebalance: Close A, increase B to 700 collateral

During step 4 validation:

- `totalAssets = 100 (vault) + 300 (from closing A) = 400`
- `totalRequired = 700 (for new Position B size)`
- Validation fails with `InsufficientVaultBalance`

However, during actual execution (step 5), closing Position A would free 300 collateral, making the total available capital 400, which is indeed sufficient for the 700 collateral requirement when combined with Position B's existing 400 collateral.

Impact

Rebalance transactions revert unnecessarily despite sufficient collateral being available through the operation's internal logic. This breaks legitimate rebalancing strategies and degrades protocol functionality.

Code Snippet

Tool Used

Manual Review

Recommendation

Close all rebalanced positions first and then do the validation. This will solve both of the problems (Ostium oracle fee / Instantly freed capital). New flow can look like the following:

1. Consolidate Safe collateral into vault
2. Process closes all of positions (both `isIncrease = false` and `isIncrease = true`)
3. Consolidate freed capital from closes
4. Validate sum of opens \leq available balance
5. Process opens (`isIncrease = true`, `amount > 0`)

Discussion

abg4

Hey @g-lightspeed, here is a PR that addresses the issue:

<https://github.com/BlendMoney/contracts/pull/102>

Issue M-2: Missing check for pending open orders allows multiple positions at different indexes [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/23>

Summary

The `_openPosition` function in `OstiumVaultController` validates that no existing position exists at index 0, but does not check for pending open market orders. If a pending open order exists and has not yet expired, calling `_openPosition` again will create another pending order. When both orders are executed by keepers, Ostium assigns indexes using `firstEmptyTradeIndex()`, causing one position to be at index 0 and the other at a different index. Since the controller only operates on index 0, it loses control over positions at other indexes, requiring manual management via Safe.

Vulnerability Detail

The `_openPosition` function checks for existing positions at index 0 using `OSTIUM_STORAGE.getOpenTrade(address(this), pairIndex, 0)` and requires `existingTrade.collateral == 0`. However, it does not check for pending open market orders using `OSTIUM_TRADING_STORAGE.pendingMarketOpenCount(address(this), pairIndex)`.

Ostium's market orders are two-step processes: the controller commits a trade and keepers execute it later. When a keeper executes a pending open order, the callback uses `storageT.firstEmptyTradeIndex(trade.trader, trade.pairIndex)` to assign the trade index, which finds the first empty slot where `openTrades[_trader][_pairIndex][i].leverage == 0`.

If `_openPosition` is called while a pending open order exists (e.g., during repeated rebalancing attempts while keepers are slow), it will create a second pending open order. When keepers execute both orders:

- The first executed order gets index 0 (the first empty slot)
- The second executed order gets index 1 or higher (the next empty slot after index 0 is taken)

The controller's invariant is that it only operates on index 0, as evidenced by all operations (open, close, position checks) hardcoding `index: 0`. Once a position is opened at a non-zero index, the controller cannot detect, close, or manage it through its normal operations, as it only checks and operates on index 0.

Impact

Multiple positions can be opened at different indexes for the same pair, violating the controller's single-position-per-pair invariant. Positions at non-zero indexes become unmanaged by the controller, requiring manual intervention via Safe to close or manage them.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/OstiumVaultController.sol#L275-L278>

Tool Used

Manual Review

Recommendation

Before creating a new pending open order, check for existing pending open orders using `OSTIUM_TRADING_STORAGE.pendingMarketOpenCount(address(this), pairIndex)`. If `pendingMarketOpenCount > 0`, either:

- Revert with a custom error indicating a pending open order exists, or
- Check if the pending orders have timed out and clear them using `OSTIUM_TRADING.openTradeMarketTimeout()` before proceeding.

Issue L-1: SwapAdapter might not work with stETH [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/12>

Summary

The `checkSwapInvariants` modifier's strict zero-balance requirement after swaps is too restrictive for certain yield-bearing tokens like stETH. Due to stETH's share-based rebasing mechanics and integer division rounding, it's impossible to guarantee exactly zero balance after a transfer, as 1-2 wei may remain due to rounding. This causes legitimate swaps to fail unnecessarily.

Vulnerability Detail

The `checkSwapInvariants` modifier contains:

```
require(primary.balanceOf(address(this)) == 0, AdapterNotEmpty());
require(secondary.balanceOf(address(this)) == 0, AdapterNotEmpty());
```

However, there's a 12 wei corner cases for some tokens like stETH:

stETH balance calculation includes integer division, and there is a common case when the whole stETH balance can't be transferred from the account while leaving the last 1-2 wei on the sender's account.

In this case, even though adapter sends all stETH to the receiver, 1-2 wei can remain at SwapAdapter contract. Due to `checkSwapInvariants` modifier, the whole transaction will revert with `AdapterNotEmpty`

Impact

SwapAdapter is not compatible with share-based rebasing tokens like stETH

Code Snippet

Tool Used

Manual Review

Recommendation

Transfer remaining amounts to recipient, instead of requiring zero remaining amount

Discussion

james-a-morris

We acknowledge this as a **known token compatibility limitation** and consider it **Low/Informational** severity.

The strict zero-balance check is an intentional safety invariant, as documented in the NatSpec (line 179): "Prevents dust or retained tokens after swap execution."

This design choice means rebasing tokens like stETH that exhibit share-based rounding behavior are not compatible with this adapter. The supported token list for any strategy is controlled by governance behind a timelock, and such tokens can be excluded from configurations.

This is a documented design tradeoff rather than an exploitable vulnerability - the worst case is a reverted transaction, not loss of funds.

james-a-morris

Merged from <https://github.com/BlendMoney/contracts/pull/118>

Issue L-2: `_validateVault` can skip validating a self-parent destination market vault [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/14>

Summary

The `_validateVault` helper in `VaultToVaultAction` returns early when `configVault == marketVault`, which means no validity check is performed for that vault's configuration. When `strategy.destinationVault == strategy.destinationMarketVault`, the call `_validateVault(strategy.destinationVault, strategy.destinationMarketVault)` will therefore accept a destination market vault without ever checking that it has a nonzero controller configured, leaving this case outside of the normal validation guarantees.

Vulnerability Detail

The `_execute` function of `VaultToVaultAction` performs three vault validations:

- `_validateVault(vault, strategy.sourceMarketVault);`
- `_validateVault(strategy.sourceVault, strategy.sourceMarketVault);`
- `_validateVault(strategy.destinationVault, strategy.destinationMarketVault);`

The last call is intended to ensure that `strategy.destinationVault` is valid in the context of `strategy.destinationMarketVault`, and that the relevant vault configuration is present and properly initialized.

However, `_validateVault` contains a short-circuit for the case where the two arguments are equal.

When `strategy.destinationVault == strategy.destinationMarketVault`, the equality branch is taken and the function returns before executing `require(address(config.controller) != address(0), InvalidVault());`. In contrast, the vault argument passed into `_execute` is validated at the `StrategyManager` level using the `validVault(vault)` modifier.

This ensures that the vault provided to the strategy has a nonzero controller in `StrategyManager`, but there is no equivalent guarantee for `strategy.destinationMarketVault` in the specific case. As a result, a destination market vault used as its own parent can bypass the intended `InvalidVault` check on its configuration in `ROLES_RECEIVER`.

Impact

In the edge case where `strategy.destinationVault == strategy.destinationMarketVault`, a destination market vault can be used without ever verifying that it has a valid, nonzero controller configured in `ROLES_RECEIVER`, weakening assumptions about destination vault correctness for vault-to-vault actions.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/VaultToVaultAction.sol#L194-L203>
<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/VaultToVaultAction.sol#L123-L129>
<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/libraries/StrategyManager.sol#L106-L109>

Tool Used

Manual Review

Recommendation

Introduce a dedicated internal helper (for example `_isValidVault(address vault)`) that checks the vault configuration in `ROLES_RECEIVER` (including `config.control != address(0)`) and call it explicitly for `strategy.destinationMarketVault` before invoking `_validateVault`. This ensures that the destination market vault is always validated for a proper configuration, even in the edge case where it is equal to `strategy.destinationVault` and `_validateVault` returns early via the self-parent branch.

Discussion

james-a-morris

The finding is technically accurate but it is our sentiment that this represents a **Low severity** issue.

The Fix

Moving the config validation check before the early return ensures all vaults are validated:

```
function _validateVault(IERC4626 configVault, IERC4626 marketVault) internal view {

    // Base Case: Validate that the vault has a config in the first place
    VaultConfig memory config = ROLES_RECEIVER.getVaultConfig(address(configVault));
    require(address(config.control) != address(0), InvalidVault());

    // Case 1: Vault is its own parent - always valid
    if (address(configVault) == address(marketVault)) {
        return;
    }

    // Case 2: Check if marketVault is whitelisted in configVault's markets
    bytes32 marketId = bytes32(uint256(uint160(address(marketVault))));
    for (uint256 i; i < config.markets.length;) {
```

```

        if (config.markets[i].marketId == marketId) return;
        unchecked {
            ++i;
        }
    }

    // Not found: validation fails
    revert InvalidVault();
}

```

Why Low Severity

1. No external attack vector - The strategyData containing destination vault addresses is set via _updateVaultConfig, which is only callable through the trusted broadcaster path in RolesReceiver. This path is controlled by governance behind a timelock.
2. Configuration issue, not code exploit - Triggering this bypass requires governance to set a malformed VaultToVaultStrategyData where destinationVault == destinationMarketVault points to an unconfigured address. An attacker cannot induce this condition.
3. Defense-in-depth improvement - The fix adds validation consistency but does not close an exploitable attack path since the strategy configuration is already protected by the timelock.

0xklapouchy

I agree that this would be a configuration error, and only then would a malicious executor (if such exist) be able to deposit to an unconfigured address.

Issue L-3: `_execute` can revert on valid same-asset swaps due to strict balance check [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/15>

Summary

The `_execute` function in `VaultToVaultAction` enforces `require(destBalanceAfter > destBalanceBefore, InvalidAmount())`; after the swap. If the source and destination assets are the same and no additional withdrawal is needed (the `if (currentBalance < extra.amount)` branch is skipped), a valid 1:1 swap that leaves the adapter's balance unchanged will cause an unintended revert.

Vulnerability Detail

In the same-asset case, the adapter can already hold the full `extra.amount` before withdrawing from the source vault. If no withdrawal is performed, `destBalanceBefore` and `destBalanceAfter` can both equal the existing balance while the swap adapter correctly returns tokens 1:1, making `destBalanceAfter == destBalanceBefore` and triggering `InvalidAmount()` even though the operation is valid.

Impact

When source and destination assets are the same and no withdrawal occurs, a valid 1:1 swap can revert with `InvalidAmount()` purely because the post-swap balance did not strictly increase.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/VaultToVaultAction.sol#L170-L172>

Tool Used

Manual Review

Recommendation

Relax the post-swap balance check in the same-asset path so it does not revert on a valid 1:1 swap that leaves the balance unchanged. For example, skip this require when `sourceAsset == destinationAsset` and rely on the swap adapter's own invariants and

slippage checks (such as `strategy.maxSlippageBps` and oracle-based quoting) to ensure that the strategy does not suffer value loss beyond acceptable bounds.

Discussion

james-a-morris

wouldn't dest balance before == dest balance after imply that all of this was a no-op?
presumably that would indicate a failure position and a valid revert

james-a-morris

and dest balance before > dest balance after means we somehow lost money (impossible)

Oxklapouchy

It is just an edge case and Info issue. We want to move 1 USDC from Vault A to Vault B, but at the same time there is 1 USDC in the user Safe.

As such, we will:

1. `uint256 destBalanceBefore = destinationAsset.balanceOf(address(this));` we store 1 USDC here.
2. We will not withdraw anything from Vault A here: `if (currentBalance < extra.amount) {}`.
3. We will perform the same swap here: `strategy.swapAdapter.swapToCollateral(a 1:1 USDC transfer)`.
4. We will recheck the balance: `uint256 destBalanceAfter = destinationAsset.balanceOf(address(this));` it will still be 1 USDC.
5. We will revert here: `require(destBalanceAfter > destBalanceBefore, InvalidAmount());`.

Issue L-4: Potential amount mismatch in VaultDepositExecuted event [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/16>

Vulnerability Detail

In `VaultToVaultAction`, `_depositAssets` uses the Safe's `balanceOf()` result as the deposited amount, but the `VaultDepositExecuted` event emits `destBalanceAfter - destBalanceBefore`. If the Safe already holds destination tokens before the deposit, the emitted amount (balance delta) can diverge from the actual deposited amount, leading to misleading off-chain accounting.

Recommendation

Update `VaultDepositExecuted` to include separate fields for the swapped amount and the deposited amount.

Issue L-5: Uncontrolled use of Safe balance for swap and destination deposit [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/17>

Vulnerability Detail

In `VaultToVaultAction._execute`, the adapter uses the entire Safe balance of source and destination assets instead of the explicitly requested `extra.amount` or the actual swap output.

After withdrawal, the code transfers the full `sourceAsset.balanceOf(address(this))` to the swap adapter, which can include pre-existing tokens. If `currentBalance > extra.amount`, the swap will exceed `extra.amount` unintentionally.

For deposits, `_depositAssets` is called with type(`uint256`).`max`, causing it to deposit the entire Safe balance of `destinationAsset`, including any pre-existing tokens, not just the swap result.

This means the caller has no strict control over swap and deposit amounts, which can cause unexpected portfolio changes.

Recommendation

Constrain the swap to `extra.amount` by computing `swapAmount = extra.amount == type(uint256).max ? balance : extra.amount` and transferring only `swapAmount` to the adapter.

For deposits, calculate `depositAmount = destBalanceAfter - destBalanceBefore` and pass this specific amount to `_depositAssets` instead of type(`uint256`).`max`. This ensures the action only moves the requested amount and the actual swap output, avoiding unintended consumption of pre-existing Safe balances.

Issue L-6: Missing @custom:reverts documentation for VaultToVaultAction._execute [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/18>

Vulnerability Detail

The abstract base `VaultActionController` requires that concrete implementations of `_execute` document their revert conditions, and other adapters use `@custom:reverts` tags to describe failure modes.

However, `VaultToVaultAction._execute` lacks any `@custom:reverts` NatSpec tags despite having multiple revert paths.

Recommendation

Add a `@custom:reverts` block to the `_execute` function NatSpec that enumerates all custom errors and their conditions.

Issue L-7: Unnecessary swap path when source and destination assets are identical [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/19>

Vulnerability Detail

In `VaultToVaultAction._execute`, the implementation always performs a token transfer to the swap adapter and calls `swapToCollateral`, even when the source and destination vaults use the same underlying ERC20 asset. When `sourceAsset == destinationAsset`, this swap is logically a no-op: the controller transfers the same token to an external adapter and back, introducing two unnecessary token transfers.

Recommendation

Add a short-circuit path that checks if `(address(sourceAsset) == address(destinationAsset))` before performing the swap. In this branch, skip the `safeTransfer` to the swap adapter and the `swapToCollateral` call entirely, and directly proceed to deposit the tokens into the destination vault using the known amount.

Issue L-8: executeRebalance validates against total vault assets instead of user's share [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/20>

Summary

The executeRebalance function in BasePerpVaultController validates rebalance amounts against `totalAssets()`, which returns the entire vault's assets. However, the user executing via `delegatecall` only owns shares in the vault, not all assets. This causes the validation to pass incorrectly when the user lacks sufficient assets, leading to failures later during actual token transfers in `_openPosition`.

Vulnerability Detail

The code calculates `totalAvailable = IERC4626(vault).totalAssets()` and then validates that the sum of rebalance amounts is less than or equal to this value. The `totalAssets()` function returns the total assets held by the entire vault, not the assets attributable to the caller's shares.

Since `executeRebalance` is executed via `delegatecall` from a user's Safe, `address(this)` refers to the Safe. The Safe only owns a portion of the vault's shares, so it can only withdraw a corresponding portion of the assets. The validation incorrectly uses the vault's total assets, which will almost always be larger than what the Safe can actually withdraw, causing the check to pass even when the Safe lacks sufficient assets.

When `_openPosition` attempts to withdraw the required amount later, it will fail with insufficient balance errors, but only after the validation has already passed and potentially after other operations have been executed.

Impact

The validation check is ineffective and will pass in cases where the user lacks sufficient assets, leading to transaction failures during actual withdrawals in `_openPosition` and wasted gas from partial execution.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/libraries/BasePerpVaultController.sol#L178>
<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/libraries/BasePerpVaultController.sol#L185>

Tool Used

Manual Review

Recommendation

Replace `totalAssets()` with `maxWithdraw(address(this))` to get the maximum amount the Safe can withdraw based on its shares:

```
-     uint256 totalAvailable = IERC4626(vault).totalAssets();  
+     uint256 totalAvailable = IERC4626(vault).maxWithdraw(address(this));
```

Alternatively, calculate based on the Safe's shares: `IERC4626(vault).previewRedeem(IERC4626(vault).balanceOf(address(this)))`. This ensures the validation checks against the actual withdrawable amount for the caller, not the entire vault's assets.

Discussion

james-a-morris

This finding is wrong. We nix it here: <https://github.com/BlendMoney/contracts/pull/102>

Also - this is going to bite us later when we use VaultV2: <https://github.com/morpho-org/vault-v2/blob/9db88c4351808bd21148e2a03636190d1946a5d8/src/VaultV2.sol>
allowbreak #L742-L745

0xklapouchy

@james-a-morris Can you add me to this repo: <https://github.com/BlendMoney>

0xklapouchy

Also - this is going to bite us later when we use VaultV2: <https://github.com/morpho-org/vault-v2/blob/9db88c4351808bd21148e2a03636190d1946a5d8/src/VaultV2.sol>
allowbreak #L742-L745

This is why `IERC4626(vault).previewRedeem(IERC4626(vault).balanceOf(address(this)))` was also suggested.

g-lightspeed

@james-a-morris I also agree with @0xklapouchy. The following check is much safer:

```
IERC4626(vault).previewRedeem(IERC4626(vault).balanceOf(address(this)))
```

Issue L-9: Redundant SafeERC20 import and using statement in BasePerpVaultController [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/21>

Vulnerability Detail

In BasePerpVaultController, SafeERC20 is imported from OpenZeppelin and has a using SafeERC20 for IERC20; directive, but no SafeERC20 methods (safeTransfer, safeTransferFrom, safeApprove, etc.) are actually used anywhere in the contract. The contract only uses standard IERC20 methods like balanceOf() and IERC20Metadata methods like decimals().

Recommendation

Remove the SafeERC20 import, keeping only IERC20, and remove the using SafeERC20 for IERC20; statement.

Discussion

abg4

This looks like a duplicate of

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/27>

Issue L-10: Ostium contract addresses stored as immutable may become stale if registry updates [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/22>

Summary

The OstiumVaultController constructor stores contract addresses from the Ostium registry as immutable variables (OSTIUM_TRADING, OSTIUM_TRADING_STORAGE, OSTIUM_PAIRS_STORAGE, OSTIUM_OPEN_PNL). While it is highly unlikely, the Ostium registry can update these contract addresses, causing the stored immutable addresses to become stale and potentially point to outdated or incorrect contracts.

Vulnerability Detail

In the constructor, contract addresses are fetched from the registry via OSTIUM_REGISTRY.getContractAddress() and stored as immutable variables. These addresses are then used throughout the contract for all Ostium interactions. If the Ostium registry updates any of these contract addresses after deployment, the controller will continue using the old addresses stored at construction time, potentially interacting with deprecated or incorrect contracts instead of the current ones.

Impact

If the registry updates contract addresses, the controller will use stale addresses, which could lead to interactions with outdated contracts or failures if old contracts are deprecated. While this scenario is unlikely, it represents a potential risk if the registry is updated.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/OstiumVaultController.sol#L177-L180>

Tool Used

Manual Review

Recommendation

Instead of storing contract addresses as immutable, always call `OSTIUM_REGISTRY.getContractAddress()` during execution to fetch the current address. This ensures the controller always uses the latest addresses from the registry, even if they are updated.

Issue L-11: Pending market orders can accumulate and cause oracle fee waste or lock collateral [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/24>

Summary

Ostium's market orders are two-step processes: the controller commits a trade and waits for keepers to execute. Pending orders can timeout after `marketOrdersTimeout` blocks, but `OstiumVaultController` does not check for or clear timeouted pending orders before creating new ones. This can lead to multiple pending close orders accumulating (each consuming oracle fee), hitting the `maxPendingMarketOrders` limit, and wasting fees when keepers execute redundant orders. For pending open orders, locked collateral cannot be freed until keepers process, leaving no control over execution.

Vulnerability Detail

When `_closePosition()` is called, it creates a pending market close order that waits for keeper execution. If keepers are unavailable or slow, multiple calls to `_closePosition()` (e.g., during repeated rebalancing attempts) will create multiple pending close orders for the same position. Each pending close order consumes an oracle fee when created, and when keepers eventually execute, they will process all pending orders, consuming all accumulated oracle fees but only closing the position once. The redundant close orders waste fees without providing value.

Similarly, `_openPosition()` creates pending open orders that lock collateral. If keepers are unavailable, the controller has no mechanism to cancel timeouted pending open orders to free the locked collateral, even though Ostium provides `openTradeMarketTimeout()` for this purpose. The controller must wait indefinitely for keeper execution or manually call the timeout function via Safe.

Ostium enforces a `maxPendingMarketOrders` limit per trader. If keepers are down for an extended period and the controller creates multiple pending orders through repeated rebalancing attempts, it can reach this limit, blocking all further market operations until pending orders are cleared.

Ostium provides timeout functions (`openTradeMarketTimeout()` and `closeTradeMarketTimeout()`) that allow traders to cancel pending orders after `marketOrdersTimeout` blocks have passed. However, `OstiumVaultController` never checks for or clears timeouted pending orders before creating new ones, leading to accumulation and waste.

Impact

Multiple pending close orders can accumulate during keeper downtime, each consuming oracle fees. When keepers resume, they execute all pending orders, wasting fees on redundant closes. Pending open orders lock collateral with no mechanism to cancel timeouted orders, preventing capital from being freed for other uses. The controller can hit the `maxPendingMarketOrders` limit, blocking all market operations. Oracle fees are wasted on redundant operations, and collateral remains locked unnecessarily.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/OstiumVaultController.sol#L202> <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/OstiumVaultController.sol#L258> <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/libraries/BasePerpVaultController.sol#L170>
<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/libraries/BasePerpVaultController.sol#L197>

Tool Used

Manual Review

Recommendation

When calling `_closePosition()` check for timeouted pending market orders. For each timeouted order (`where block.number >= order.block + marketOrdersTimeout`), call the appropriate timeout function:

- For timeouted close orders: call `OSTIUM_TRADING.closeTradeMarketTimeout(orderId, false)` to clear them and refund oracle fees, preventing accumulation and fee waste.
- For timeouted open orders: call `OSTIUM_TRADING.openTradeMarketTimeout(orderId)` to cancel them and free locked collateral, allowing capital to be used elsewhere.

Issue L-12: Hardcoded 10% oracle fee limit blocks closing dust positions and prevents rebalancing [RE-SOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/25>

Summary

The `_closePosition` function in `OstiumVaultController` enforces a hardcoded 10% maximum oracle fee check (`maxOracleFee = (existingTrade.collateral * 1000) / 10000`). For dust positions with very small collateral, the fixed oracle fee can exceed 10% of the position value, causing the close to revert.

While positions can be potentially closed via Safe calls to Ostium, the automated rebalancing logic becomes unusable.

Vulnerability Detail

When closing a position, the code validates that the oracle fee does not exceed 10% of the position's collateral value. The oracle fee is a constant USDC value that does not scale with position size. For dust positions, this fixed fee can exceed 10% of the collateral, causing `require(fee <= maxOracleFee, ExcessiveOracleFee())` to revert.

Ostium's close and open operations are two-step processes. When rebalancing to open a new larger position (`isIncrease = true`), the executor first calls `_closePosition`, which commits the close transaction and waits for the keeper to process it. After the keeper executes the close, the executor must call `executeRebalance()` again to open the new position. However, if the existing position is dust and fails the 10% oracle fee check, the initial close call reverts, preventing the position from being closed. Since `OstiumVaultController` requires closing the existing position before opening a new one, the `openTrade` cannot be called while a position exists, the inability to close the dust position permanently blocks opening new positions for that pair, effectively locking rebalancing.

Impact

Dust positions cannot be closed through the rebalancing logic when the oracle fee exceeds 10% of their value, blocking subsequent rebalancing operations for that pair. The hardcoded 10% threshold also lacks configurability, preventing adjustment for different risk tolerances or market conditions.

Code Snippet

<https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/blob/main/contracts/src/adapters/OstiumVaultController.sol#L221-L223>

Tool Used

Manual Review

Recommendation

Make the maximum oracle fee threshold configurable via vault configuration instead of hardcoding 10%. Additionally, add a mechanism to force-close positions even when the fee exceeds the threshold, such as a flag in the rebalance data or a separate emergency close function.

Issue L-13: Missing oracle fee validation in `_openPosition` [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/26>

Vulnerability Detail

The `_closePosition` function in `OstiumVaultController` validates that the oracle fee does not exceed 10% of the position's collateral value before closing. However, `_openPosition` lacks this same validation when opening new positions. This inconsistency means positions can be opened with collateral amounts where the oracle fee would exceed 10% of the position value, but the same positions cannot be closed later through the rebalancing logic due to the fee check. The oracle fee is a constant USDC value that does not scale with position size, so small positions opened without validation can become uncloseable.

Recommendation

Consider adding similar oracle fee validation to `_openPosition` that exists in `_closePosition`.

Issue L-14: Rounding down in PriceLib weakens slippage protection in SwapAdapter [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/28>

Vulnerability Detail

All calculations in PriceLib round down: `applySlippage()` uses integer division, and `quoteToBase()` and `baseToQuote()` use `mulDiv()` which also rounds down. In SwapAdapter, these functions are chained when calculating `amountOutMin` for slippage protection. For example, in `swapToCollateral()`: `expectedCollateral = PriceLib.quoteToBase(amount, oracle)` (rounds down), then `amountOutMin = PriceLib.applySlippage(expectedCollateral, slippageBps)` (rounds down again). This double rounding down makes `amountOutMin` smaller than the mathematically correct value, weakening the slippage check and allowing broader acceptance of swaps that may exceed the intended slippage tolerance.

Recommendation

Either acknowledge this behavior in code comments, or modify the slippage calculation to use rounding up when computing `amountOutMin` to make the check stricter.

Issue L-15: Unnecessary max approval in MorphoVaultLib._depositAssets [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/29>

Vulnerability Detail

In MorphoVaultLib._depositAssets, the function approves type(uint256).max to the vault before calling vault.deposit(assets, address(this)), even though only assets amount is needed. This grants unlimited approval to the vault, which is unnecessary and increases risk. If the vault contract is compromised or has a bug, it could potentially transfer more tokens than intended. The latter slippage check only validates the share price ratio, not that the vault consumed exactly the approved amount, so approving only the required assets amount would provide better protection.

Recommendation

Change the approval to only approve the exact assets amount: SafeERC20.forceApprove(underlying, address(vault), assets).

Issue L-16: Missing @custom:reverts documentation for SwapAdapter swap functions [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/30>

Vulnerability Detail

The `swapToCollateral` and `swapToLoanToken` functions in `SwapAdapter` only document `InvalidRecipient` in their `@custom:reverts` tags, but both functions have multiple additional revert conditions that are not documented. These include `ZeroAmount` when the input token balance is zero or when expected output is zero, `MaxSlippageInvalid` when `maxSlippageBps` exceeds `MAX_SLIPPAGE_BPS`, `SlippageExceedsMaximum` when `swapParams.slippageBps` exceeds `strategyConfig.maxSlippageBps`, and `SlippageExceeded` when the actual swap output is less than the minimum required amount.

Recommendation

Add a complete `@custom:reverts` block to both `swapToCollateral` and `swapToLoanToken` that enumerates all custom errors and their conditions.

Issue L-17: Missing security warning about token approvals and privileges for SwapAdapter and BalanceReplacementAdapter [RESOLVED]

Source: <https://github.com/sherlock-audit/2026-01-blend-money-jan-12th/issues/31>

Vulnerability Detail

Both SwapAdapter and BalanceReplacementAdapter can execute arbitrary calls through the Bundler3 multicall mechanism (`_executeSwapCalls` and `target.call` respectively). However, neither contract documents in their NatSpec that users should never grant token allowances, approvals, or any other privilege access to these contracts. If a user approves tokens to these contracts, an attacker could craft malicious swap calls that use `transferFrom` to drain the approved tokens. Similarly, any other privileges granted to these contracts could be exploited through arbitrary call execution, leading to fund loss.

Recommendation

Add explicit security warnings to the contract-level NatSpec (`@notice` or `@dev`) for both SwapAdapter and BalanceReplacementAdapter stating that users must never approve tokens or grant any privileges to these contracts, as they execute arbitrary calls and any approvals will be at risk of being drained through malicious swap call data.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.