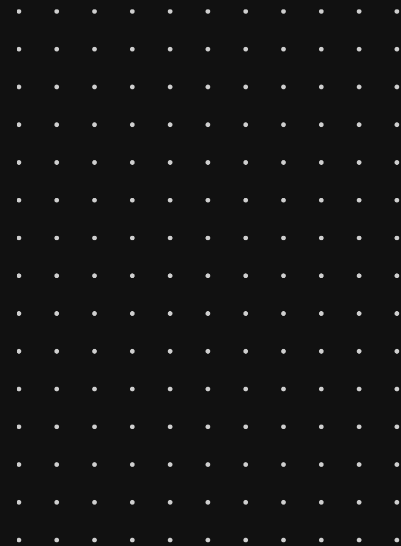
cecuro

Audit Report

March 3, 2026



PROJECT

Zyfai

Audit Overview

Project: zyfai
Repository: https://github.com/ondefy/erc7540-wrapper
Audit Date: March 3, 2026
Commit: a1f65cf4
Scope: 4 files

Security audit completed.

Audit Scope

The following 4 files were included in this security audit:

- `src/ISemiAsyncRedeemVault.sol`
- `src/SmartAccountProxy.sol`
- `src/SemiAsyncRedeemVault.sol`
- `src/SmartAccountWrapper.sol`

Findings

HIGH

Partial requestWithdraw/requestRedeem can burn all shares but drop the remaining asset shortfall (rounding bug)

Locations:

`src/SemiAsyncRedeemVault.sol:324-342`

`src/SemiAsyncRedeemVault.sol:279-322`

Description:

`_processRequest()` attempts to satisfy a withdraw/redeem immediately from `idleAssets()` and create an async request for any shortfall.

However, it computes the shares to burn for the immediate portion as:

- `sharesToRedeem = _convertToShares(assetsToWithdraw, Math.Rounding.Ceil)`
- `sharesToRequest = shares - sharesToRedeem`

and only creates a pending request when **both** `assetsToRequest > 0` **and** `sharesToRequest > 0`.

Because `sharesToRedeem` is rounded **up**, it can equal the *entire* `shares` even when `assetsToWithdraw < assets`. In that case:

- the user's full share amount is burned in the immediate `_withdraw`
- the user receives only `assetsToWithdraw`
- **no pending request is created for `assetsToRequest`** (`sharesToRequest == 0`), permanently dropping the shortfall

This is a direct loss of user value and can create unowned/stuck assets or transfer value to remaining shareholders.

Vulnerable Code:

```
function _processRequest(uint256 assets, uint256 shares, address controller, address receiver,
address owner)
 internal
 returns (bytes32)
{
 uint256 maxAssets = maxWithdraw(owner);
 uint256 assetsToWithdraw = Math.min(assets, maxAssets);
 uint256 assetsToRequest = assets - assetsToWithdraw;

 uint256 sharesToRedeem = _convertToShares(assetsToWithdraw, Math.Rounding.Ceil);
 uint256 sharesToRequest = shares - sharesToRedeem;

 if (assetsToWithdraw > 0) _withdraw(_msgSender(), receiver, owner, assetsToWithdraw,
sharesToRedeem);

 if (assetsToRequest > 0 && sharesToRequest > 0) {
 return _requestWithdraw(_msgSender(), controller, receiver, owner, assetsToRequest,
sharesToRequest);
 }
 return bytes32(0);
}
```

Concrete Exploit / Loss Example (with OZ ERC4626 math):

OpenZeppelin's ERC4626 uses virtual shares/assets:

- `previewRedeem(shares) = floor(shares * (totalAssets+1) / (totalSupply+1))`
- `_convertToShares(assets, Ceil) = ceil(assets * (totalSupply+1) / (totalAssets+1))`

Assume:

- `totalSupply = 1` share (victim)
- `totalAssets = 1000` assets (mostly allocated, not idle)
- `idleAssets()` allows only `assetsToWithdraw = 250`

Victim calls `requestWithdraw(maxRequestWithdraw(victim))`:

- `maxRequestWithdraw = convertToAssets(1) = floor(1 * (1000+1) / (1+1)) = 500`
- `shares = previewWithdraw(500) = ceil(500 * (1+1) / (1000+1)) = ceil(1000/1001) = 1`

Inside `_processRequest`:

- `assetsToWithdraw = 250`, so `assetsToRequest = 250`
- `sharesToRedeem = ceil(250 * (1+1) / (1000+1)) = ceil(500/1001) = 1`
- `sharesToRequest = 1 - 1 = 0`

Result:

- `_withdraw(..., assetsToWithdraw=250, sharesToRedeem=1)` burns the user's **entire** share and pays only 250 assets
- since `sharesToRequest == 0`, the contract creates **no pending request** for the remaining 250 assets

The user permanently loses the right to the shortfall assets.

Impact:

- **Permanent user fund loss / value loss** during partial liquidity situations (the core semi-async scenario).
- Can create **unowned assets** (if all shares are burned but not all assets are owed via obligations).
- Enables value transfer to remaining shareholders, and can be combined with share-price manipulation / liquidity throttling to steal value from withdrawers.

Recommendation:

Refactor the split logic so that a nonzero `assetsToRequest` can never occur with a zero `sharesToRequest`. Typical fixes include:

- compute the immediate portion in **shares** first (bounded by `maxRedeem`), then derive `assetsToWithdraw`, leaving the remaining shares as pending; or
- ensure rounding for `sharesToRedeem` cannot round up into consuming the entire `shares` unless `assetsToWithdraw == assets`.

Also add invariant checks and revert if a shortfall would be dropped.

MEDIUM

SmartAccountWrapper allows `smartAccount = address(this)`, breaking accounting and enabling depositor-to-depositor value theft via mispriced shares

Locations:

`src/SmartAccountWrapper.sol:118-123`

`src/SmartAccountWrapper.sol:176-180`

`src/SmartAccountWrapper.sol:186-188`

Description:

`SmartAccountWrapper` does not prevent setting `smartAccount` to `address(this)` (either in `initialize` or via `setSmartAccount`). If `smartAccount == address(this)`, `_transferToSmartAccount`:

- increments `allocatedAssets` **as if assets left the vault**, but
- performs `safeTransfer` to itself (no net token movement), leaving the tokens in the wrapper balance.

This causes `totalAssets()` to **double count** the same assets (`vaultBalance + allocatedAssets`) and corrupts ERC4626 share pricing.

Because ERC4626 share issuance uses `totalAssets()` in its conversion formulas, this mispricing can be exploited by early depositors to withdraw more underlying than they contributed, stealing from later depositors.

Evidence:

```
function _transferToSmartAccount(uint256 assets) internal {
 address _smartAccount = smartAccount();
 if (_smartAccount == address(0)) revert SA__SmartAccountNotSet();
 _getSmartAccountWrapperStorage().allocatedAssets += assets;
 IERC20(asset()).safeTransfer(_smartAccount, assets);
}

function setSmartAccount(address smartAccount_) public onlyOwner {
 if (smartAccount_ == address(0)) revert SA__ZeroAddress();
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
}
```

No check exists to forbid `smartAccount_ == address(this)`.

Exploit Sketch (Concrete Numbers):

Assume `smartAccount` misconfigured to `address(this)`. 1) Attacker deposits 100 underlying.

- Wrapper balance = 100
- `allocatedAssets += 100` (now 100)
- transfer to self leaves balance = 100
- `totalAssets = balance(100) + allocated(100) = 200`

2) Victim deposits 100 underlying.

- Shares minted are diluted because `previewDeposit` uses inflated `totalAssets`.

3) Attacker withdraws/redeems at the inflated share price, draining a disproportionate share of the wrapper's real underlying balance (which includes the victim's deposit).

Impact:

- Breaks ERC4626 accounting invariants and can lead to significant depositor fund loss.
- Also risks permanent withdrawal/claimability issues because `allocatedAssets` no longer reflects assets outside the wrapper.

Recommendation:

Explicitly forbid `smartAccount` being set to `address(this)` (and consider forbidding other nonsensical values). Additionally, consider asserting or sanity-checking that changes to `allocatedAssets` correspond to actual token movements (e.g., via balance-delta accounting) to prevent silent divergence.

MEDIUM

Any user with shares can grief/DoS owner and smartAccount accounting updates by creating pending withdrawals (pendingWithdrawals() gate)

Locations:

`src/SmartAccountWrapper.sol:129-158`

`src/SemiAsyncRedeemVault.sol:165-178`

`src/SemiAsyncRedeemVault.sol:308-341`

Description:

`SmartAccountWrapper` gates key privileged accounting update functions on `pendingWithdrawals() == 0`:

- `forceTransmitAllocatedAssets()` (onlyOwner)
- `transmitAllocatedAssets()` (onlySmartAccount)

But `pendingWithdrawals()` is derived from user-controlled redeem requests (`requestRedeem` / `_requestWithdraw`). Any user with shares can create a (possibly tiny) async redeem request that makes `pendingWithdrawals() > 0`, causing these privileged functions to revert.

This is an access-control availability issue (focus area #10): a non-privileged user can block governance/admin operations whose `require` conditions depend on user-manipulable state.

Evidence:

Privileged functions revert when pending withdrawals exist:

```
function forceTransmitAllocatedAssets(uint256 assets) public onlyOwner {
 if (pendingWithdrawals() > 0) revert SA__PendingWithdrawals();
 ...
}

function transmitAllocatedAssets(uint256 assets) public onlySmartAccount {
 if (pendingWithdrawals() > 0) revert SA__PendingWithdrawals();
 ...
}
```

Users can create pending withdrawals via requestRedeem:

```
function requestRedeem(uint256 shares, address controller, address owner) public virtual
returns (uint256) {
 ...
 bytes32 withdrawKey = _processRequest(assets, shares, controller, controller, owner);
 return uint256(withdrawKey);
}
```

If the vault holds little/no on-hand balance (common here since deposits are forwarded to the smart account), `_processRequest` will create an async request (`_requestWithdraw`) and increase outstanding obligations.

pendingWithdrawals() depends on outstanding obligations:

```
uint256 outstanding = _outstandingObligations();
...
uint256 processed = vaultBalance + claimableFromStrategies() + pendingDeallocationAssets();
if (processed >= outstanding) return 0;
return outstanding - processed;
```

In `SmartAccountWrapper`, `claimableFromStrategies()` and `pendingDeallocationAssets()` are always 0, so `processed` is typically just `vaultBalance`.

Impact:

A user can keep `pendingWithdrawals() > 0` by creating/maintaining outstanding redemption obligations, which prevents:

- The owner from using `forceTransmitAllocatedAssets` (cannot correct accounting upward).
- The smart account from calling `transmitAllocatedAssets` (cannot publish updated allocation state).

This can disrupt admin operations and any downstream integrations relying on timely accounting updates.

Recommendation:

Avoid making privileged maintenance actions uncallable based solely on user-controlled state. Options include:

- Allowing these functions even when pending withdrawals exist, or
- Providing a separate emergency/admin override, or
- Tightening the condition to only forbid changes that would violate invariants, rather than blanket `pendingWithdrawals()>0`.

MEDIUM

Claims can become permanently impossible if `asset.transfer` to the stored receiver reverts (blacklist/pause), with no ability to redirect payout

Locations:

`src/SemiAsyncRedeemVault.sol:407-434`

`src/SemiAsyncRedeemVault.sol:373-382`

Description:

Each async request permanently stores a `receiver` address in `WithdrawRequest`. When `claim()` is executed, assets are always transferred to that stored receiver:

```
IERC20(asset()).safeTransfer(request.receiver, amount);
```

If the underlying asset implements transfer restrictions (e.g., USDC blacklist / pause), or any other condition makes `transfer(receiver, amount)` revert, then:

- the request can remain **forever unclaimable in practice** for that receiver (blacklist is typically permanent)
- there is **no on-chain mechanism** to update the receiver or re-route the payout to a new address

This creates a stuck-state risk for individual requests (and potentially protocol-wide during pauses), even after they become “claimable” by accounting.

Evidence:

Receiver is stored on request creation:

```
$.withdrawRequests[withdrawKey] = WithdrawRequest({
 ...
 receiver: receiver,
 ...
});
```

Claim always transfers to stored receiver:

```
function claim(bytes32 withdrawKey) public returns (uint256) {
 ...
 IERC20(asset()).safeTransfer(request.receiver, amount);
}
```

Impact:

High: funds for a request can be effectively locked indefinitely if the receiver becomes unable to receive the asset (blacklisted / sanctioned / permanently blocked). This is especially realistic for assets like USDC.

Recommendation:

Provide a bounded recovery mechanism to change the receiver (e.g., by owner/controller authorization), or a safe alternative claim path (e.g., claim to controller) when transfers to the receiver fail.

MEDIUM

Redeem requests are converted into fixed-asset debt at request time (requestedAssets), allowing loss-avoidance and potentially insolvent/stuck queues after strategy losses

Locations:

`src/SemiAsyncRedeemVault.sol:308-322`

`src/SemiAsyncRedeemVault.sol:350-390`

`src/SemiAsyncRedeemVault.sol:422-433`

Description:

`requestRedeem(shares, ...)` computes an **asset amount immediately** via `previewRedeem(shares)` and records that value as `requestedAssets` in the withdrawal request. The eventual settlement (`claim`) pays **exactly `requestedAssets`**, independent of how `totalAssets()` evolves between request and claim.

This turns a share-denominated redemption request into a **fixed-asset IOU priced at request time**, rather than a claim on vault assets at the exchange rate when the request becomes Claimable/Claimed.

Evidence:

- Asset amount is fixed at request time:

```
uint256 assets = previewRedeem(shares);
bytes32 withdrawKey = _processRequest(assets, shares, controller, controller, owner);
```

- Request stores both `requestedShares` and `requestedAssets`, but claim pays `requestedAssets`:

```
$.withdrawRequests[withdrawKey] = WithdrawRequest({
 requestedAssets: assetsToRequest,
 ...
 requestedShares: sharesToRequest,
 ...
});

uint256 amount = request.requestedAssets;
IERC20(asset()).safeTransfer(request.receiver, amount);
```

Why this is a business-logic risk:

Economically, the requestor exits the equity exposure immediately (shares burned) but preserves an **asset-denominated entitlement** that does not adjust to subsequent gains/losses.

This creates adverse incentives and can be exploited:

- **Loss avoidance / value transfer:** A user can request redemption before an anticipated loss; if the vault later loses assets, the request remains for the pre-loss asset amount, pushing the entire loss onto remaining shareholders.
- **Insolvency / stuck queue:** If strategy losses reduce the vault's actual assets below the sum of fixed `requestedAssets`, some requests may never become claimable, permanently locking redemptions.

Impact:

High impact if the vault's asset value can change while requests are pending (which is typical when strategies are involved). It can systematically transfer value from remaining shareholders to requesters and can lead to permanently stuck redemption queues after losses.

Recommendation:

If the intent is ERC-7540-style redeem requests, avoid locking a fixed `requestedAssets` at request time; instead, treat requests as share claims that settle at claim time (or at a defined "claimable" exchange rate), and ensure the claim path uses the recorded `requestedShares` consistently.

LOW

Authorization checks for creating async requests are controller-centric and liquidity-dependent, breaking expected controller/owner semantics

Locations:

`src/SemiAsyncRedeemVault.sol:324-342`

`src/SemiAsyncRedeemVault.sol:344-365`

`src/SemiAsyncRedeemVault.sol:349-390`

`src/SemiAsyncRedeemVault.sol:279-292`

Description:

The permission model for creating an asynchronous shortfall request is inconsistent with the apparent intent of `controller` / `owner` separation and is applied **only** when a shortfall exists.

- `_requestWithdraw` enforces `caller == controller || isOperator(controller, caller)`:

```
if (!_isAuthorized(caller, controller)) revert SA__NotAuthorized();
```

- But `_processRequest` performs the **immediate** `_withdraw` first and only enters `_requestWithdraw` for the shortfall:

```
 if (assetsToWithdraw > 0) _withdraw(_msgSender(), receiver, owner, assetsToWithdraw,
sharesToRedeem);
 if (assetsToRequest > 0 && sharesToRequest > 0) {
 return _requestWithdraw(_msgSender(), controller, receiver, owner, assetsToRequest,
sharesToRequest);
 }
```

This creates liquidity-dependent authorization behavior:

- If the vault has enough idle liquidity (`assetsToRequest == 0`), **any** caller with share allowance can route assets to an arbitrary `controller`/`receiver` via the immediate `_withdraw` path.
- If the vault has a shortfall, the **same call** reverts unless the caller is the controller (or controller's operator).

Additionally, the operator system is scoped to the **controller** (`operators[controller][op]`), but shares are taken from the **owner**. Even when the caller is an approved operator, `_requestWithdraw` still requires ERC-20 share allowance when `caller != owner`:

```
if (caller != owner) {
 _spendAllowance(owner, caller, sharesToRequest);
}
```

This undermines the semantic purpose of "operator" approvals as a bypass for ERC-20 allowance-based custody delegation (as described in ERC-7540).

The deprecated `requestWithdraw(assets, receiver, owner)` is also affected because it sets `controller = receiver`, making shortfall behavior unexpectedly depend on whether the `receiver` has authorized the caller.

Impact:

- Integrations/users can observe confusing behavior where the **same** `requestRedeem`/`requestWithdraw` call succeeds or reverts depending on idle liquidity.
- Owners cannot reliably create requests for third-party controllers/receivers unless those controllers have pre-approved the caller, defeating common "owner submits request, custodian is controller" flows.
- Operator approvals do not provide the expected delegation power (still require ERC-20 allowance), which can break assumptions in off-chain/accounting systems.

Recommendation:

Clarify and align the authorization model with intended semantics:

- Decide whether `controller` must always be the caller (or caller must be owner/operator-of-owner), and enforce it consistently regardless of liquidity.
- If operator approvals are intended to bypass ERC-20 allowances, incorporate operator logic into the share-taking path (or document that allowances are always required).

LOW

Contract claims IERC7540Redeem support but violates key ERC-7540 semantics (requestId rules, preview reverts, and no claim via withdraw/redeem)

Locations:

`src/ISemiAsyncRedeemVault.sol:8-14`

`src/SemiAsyncRedeemVault.sol:275-342`

`src/SemiAsyncRedeemVault.sol:384-387`

`src/SmartAccountWrapper.sol:215-218`

Description:

`ISemiAsyncRedeemVault` and `SemiAsyncRedeemVault` present themselves as implementing ERC-7540 asynchronous redeem (via `IERC7540Redeem`), but the actual behavior diverges materially from the ERC-7540 Final spec (EIP-7540).

Key mismatches:

1) Short-circuiting the Claim step (forbidden by ERC-7540):

ERC-7540 requires that Requests **MUST NOT** skip/short-circuit the Claim state and vaults must not “push” tokens to users after a Request. However, `requestRedeem` (and deprecated `requestWithdraw`) immediately transfers up to idle liquidity inside `_processRequest`:

```
if (assetsToWithdraw > 0) _withdraw(...); // transfers assets immediately
```

This violates the required request’(pending/claimable)’explicit claim flow.

2) `previewRedeem` / `previewWithdraw` do not revert:

ERC-7540 mandates `previewRedeem` and `previewWithdraw` **MUST revert for all callers and inputs** for async redemption vaults. This implementation inherits OZ ERC4626 previews without overriding.

3) `requestId == 0` rule is violated:

ERC-7540 states: *"If a Vault returns 0 for the requestId of any request, it MUST return 0 for all requests."* This contract returns `0` when it decides the redemption is fully satisfied immediately, and returns a nonzero hash-based id when it creates a pending request:

```
return uint256(withdrawKey); // withdrawKey is 0 or keccak-derived
```

This mixes 0 and nonzero requestIds, contrary to the standard.

4) No ERC-7540 claim via ERC-4626 `withdraw`/`redeem`:

ERC-7540's claim step for redemptions is done via the ERC-4626 `withdraw`/`redeem` methods after `requestRedeem` has already moved shares out of the owner. Here, shares are burned in `_requestWithdraw`, and later assets are claimed via a **non-standard** `claim(bytes32 withdrawKey)` method. Meanwhile, ERC-4626 `withdraw`/`redeem` remain standard and still burn shares, so they cannot be used as ERC-7540 claim functions. Integrations that implement ERC-7540 correctly (requestRedeem ' later withdraw/redeem as claim) will fail.

5) `RedeemRequest` event uses non-standard units:

ERC-7540 defines `RedeemRequest` to log the number of `shares` locked. This implementation emits `RedeemRequest(..., assetsToRequest)` (assets, not shares):

```
emit RedeemRequest(controller, owner, uint256(withdrawKey), caller, assetsToRequest);
```

6) ERC-165 signalling is misleading:

`SmartAccountWrapper.supportsInterface` returns true for `IERC7540Redeem` / `IERC7540Operator`, signalling standard compliance to integrators, while the above semantic requirements are not met.

Impact:

- ERC-7540-aware integrators/indexers can mis-handle this vault:
  - expecting previews to revert (or relying on revert behavior) but they don't,
  - expecting `requestId` semantics (especially the `0` rule) but they differ,

  - expecting to claim via ERC-4626 `withdraw`/`redeem` but the vault requires a custom `claim(bytes32)`.
- This can lead to stuck flows (protocol integrations cannot complete redemption), incorrect accounting, and broken UX.

Recommendation:

If ERC-7540 compatibility is intended, align behavior with EIP-7540: enforce preview reverts, follow requestId==0 rule, disallow push/short-circuiting, emit correct units in events, and implement the claim step via the ERC-4626 withdraw/redeem semantics as specified.

LOW

Deprecated requestWithdraw uses receiver as ERC-7540 controller, causing unexpected authorization reverts and preventing partial withdrawals to third-party receivers

Locations:

`src/SemiAsyncRedeemVault.sol:279-292`

`src/SemiAsyncRedeemVault.sol:324-342`

`src/SemiAsyncRedeemVault.sol:349-363`

Description:

`requestWithdraw(uint256 assets, address receiver, address owner)` is documented as a backwards-compatible way to request a withdrawal to an arbitrary `receiver`.

In implementation, it sets `controller = receiver` for the ERC-7540 authorization model:

```
// For backward compat: receiver serves as both controller and receiver
return _processRequest(assets, shares, receiver, receiver, owner);
```

(`src/SemiAsyncRedeemVault.sol:290-292`)

If the withdrawal cannot be fully satisfied immediately (`assetsToRequest > 0`), `_processRequest` attempts to create an async request via `_requestWithdraw(...)`, which enforces:

```
if (!_isAuthorized(caller, controller)) revert SA__NotAuthorized();
```

(`src/SemiAsyncRedeemVault.sol:358-360`)

Because `controller == receiver`, a caller who is **not** the `receiver` (and not an operator approved by `receiver`) cannot create the async shortfall request. This makes the entire `requestWithdraw` call revert when idle liquidity is insufficient, even though it could have sent a partial immediate withdrawal to `receiver`.

Impact:

- Users attempting to withdraw to a third-party `receiver` will unexpectedly revert whenever the vault cannot fully satisfy the withdrawal immediately.
- This breaks the function's documented ERC-4626-style expectation (where `receiver` is just a destination) and can cause DoS for integrations that rely on `requestWithdraw` during low-liquidity periods.

Recommendation:

If `requestWithdraw` is intended to mimic ERC-4626 semantics, do not couple `receiver` to the ERC-7540 `controller`. Consider using `controller = caller` (or a dedicated parameter) so that authorization is based on the caller/owner relationship rather than the `receiver` address.

LOW

SmartAccountWrapper share pricing relies on manually-reported `allocatedAssets`, enabling mispricing/double-counting and breaking rebasing/fee-on-transfer assumptions

Locations:

`src/SmartAccountWrapper.sol:96-173`

`src/SemiAsyncRedeemVault.sol:143-151`

Description:

`SmartAccountWrapper` overrides `allocatedAssets()` to return a storage variable that is **manually updated** via `transmitAllocatedAssets`, `transmitDeallocatedAssets`, or owner-only `forceTransmit*` functions.

`SemiAsyncRedeemVault.totalAssets()` then uses this value for ERC4626 share pricing:

```
uint256 gross = vaultBalance + allocatedAssets() + pendingDeallocationAssets() + claimable-
FromStrategies();
uint256 obligations = _outstandingObligations();
return gross - obligations;
```

In `SmartAccountWrapper`, deposits immediately transfer assets out to `smartAccount` and increment `allocatedAssets` by the nominal `assets` amount:

```
_getSmartAccountWrapperStorage().allocatedAssets += assets;
IERC20(asset()).safeTransfer(_smartAccount, assets);
```

This design has several correctness/safety pitfalls:

1) **Mispricing from stale or incorrect reporting**: `allocatedAssets` can drift from the smart account's real balance (yield, losses, manual transfers, slashing, etc.). ERC4626 conversions (`previewDeposit`, `previewRedeem`, etc.) will then mint/burn shares at an incorrect exchange rate.

2) **Double counting during deallocation**: if the smart account transfers tokens back to the wrapper (increasing `vaultBalance`) but `allocatedAssets` is not reduced in the same transaction, `totalAssets()` temporarily counts the same tokens twice (-`vaultBalance` + `allocatedAssets`), inflating share price and affecting request/preview calculations.

3) **Rebasing / fee-on-transfer incompatibility**: if the underlying token rebases (supply/balances change automatically) or charges fees on transfers, then `allocatedAssets += assets` and later manual updates may not match actual balances. This can lead to insolvency conditions (e.g., `totalAssets()` reverting with `SA__TotalAssetsUnderflow()`), incorrect share accounting, and broken withdraw/request flows.

4) **Privileged manipulation risk**: the owner can arbitrarily set `allocatedAssets` via `forceTransmitAllocatedAssets` / `forceTransmitDeallocatedAssets` (no timelock), directly manipulating `totalAssets()` and therefore the ERC4626 exchange rate.

Impact:

- Incorrect ERC4626 share pricing (users receive too many/few shares) and potential value transfer between users.
- Temporary/permanent DoS of core ERC4626 views and flows if `obligations > gross` triggers `SA__TotalAssetsUnderflow()`.
- Incompatibility with rebasing/fee-on-transfer tokens.
- Centralization risk: owner can manipulate accounting/exchange rate.

Recommendation:

- If this wrapper is intended to be generic, explicitly restrict supported assets (non-rebasing, no transfer fees) and document the trust model.
- Consider deriving `allocatedAssets` from on-chain observable balances (e.g., `IERC20(asset()).balanceOf(smartAccount)`) or enforce an on-chain proof.
- If manual reporting is required, add stronger invariants and/or timelocks/guards on owner overrides, and update `allocatedAssets` atomically with any token movement back to the wrapper to avoid double-counting windows.

LOW

Withdraw/claim fulfillment can get permanently stuck because wrapper does not (and cannot) pull assets back from smartAccount

Locations:

`src/SmartAccountWrapper.sol:112-123`

`src/SmartAccountWrapper.sol:160-167`

`src/SemiAsyncRedeemVault.sol:407-434`

`src/SemiAsyncRedeemVault.sol:441-445`

Description:

`SmartAccountWrapper` immediately forwards deposited assets to an external `smartAccount` address, leaving the wrapper with ~0 on-hand balance. The async redeem/claimability logic in `SemiAsyncRedeemVault` only considers assets **held by the wrapper** (`IERC20(asset()).balanceOf(address(this))`) plus `claimableFromStrategies()` when deciding if a request is claimable and when executing `claim()`.

In this repository's concrete implementation:

- `claimableFromStrategies()` is hardcoded to `0`
- there is **no mechanism** in the wrapper or base vault to pull funds from `smartAccount`
- `transmitDeallocatedAssets()` only updates an accounting number and does not move tokens

As a result, the entire request'claim state machine depends on an off-chain process and an external account voluntarily transferring tokens back to the wrapper. If that external step fails (smart account paused/blacklisted, operator down, key loss, etc.), users' shares can be burned via `requestWithdraw/requestRedeem` but their assets can remain indefinitely unclaimable.

Evidence:

Deposits are forwarded out of the wrapper:

```
function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal
override {
 super._deposit(caller, receiver, assets, shares);
 // transfer assets to smart account
 _transferToSmartAccount(assets);
}

function _transferToSmartAccount(uint256 assets) internal {
 _getSmartAccountWrapperStorage().allocatedAssets += assets;
 IERC20(asset()).safeTransfer(_smartAccount, assets);
}
```

Deallocation "transmit" does not actually transmit tokens:

```
function transmitDeallocatedAssets(uint256 remainingAllocatedAssets) public onlySmartAccount
{
 uint256 current = allocatedAssets();
 if (remainingAllocatedAssets > current) revert ...;
 _getSmartAccountWrapperStorage().allocatedAssets = remainingAllocatedAssets;
}
```

Claimability and claims only use the wrapper's token balance:

```
return cumulativeClaimedAssets() + IERC20(asset()).balanceOf(address(this)) + claimableFrom-
Strategies()
 >= request.cumulativeRequestedWithdrawalAssets;

IERC20(asset()).safeTransfer(request.receiver, amount);
```

Impact:

High risk of **permanent funds lock** for all users if the off-chain/operator/smartAccount leg of the workflow fails.

- Users burn shares when requesting async redemption.
- Requests may never become claimable.
- There is no bounded, on-chain recovery path to progress state.

Recommendation:

Add an on-chain, bounded fulfillment mechanism that can actually move assets back into the wrapper (or a clearly defined, permissioned settlement path), plus an emergency recovery path for cases where the smart account cannot cooperate.

LOW

ERC-4626 compliance/semantic issue: totalAssets() can revert (SA__TotalAssetsUnderflow), breaking integrator assumptions and potentially bricking core flows

Locations:

`src/SemiAsyncRedeemVault.sol:143-151`

`src/SemiAsyncRedeemVault.sol:92-99`

Description:

`SemiAsyncRedeemVault.totalAssets()` can revert with `SA__TotalAssetsUnderflow()` when `_outstandingObligations()` exceeds `gross`:

```
uint256 gross = vaultBalance + allocatedAssets() + pendingDeallocationAssets() + claimable-
FromStrategies();
uint256 obligations = _outstandingObligations();
if (obligations > gross) revert SA__TotalAssetsUnderflow();
return gross - obligations;
```

Under ERC-4626 / ERC-7575 semantics, `totalAssets()` is expected to be a non-reverting view that returns the total managed assets (even if 0 in adverse states). Reverting here can break:

- `convertToShares` / `convertToAssets` and all preview functions (they call `totalAssets()` internally),
- deposits/withdrawals that compute shares/assets via previews.

This revert can be triggered by realistic operational drift in this codebase (e.g., `allocatedAssets` being manually reported too low, losses in the smart account, or mismatch between outstanding request accounting and reported gross assets).

Impact:

- Sudden DoS of ERC-4626 core views and flows for integrators/UI when the vault is mis-accounted or becomes insolvent.
- Breaks “MUST NOT revert” expectations from ERC-4626/7575 docs that many integrations rely on.

Recommendation:

Consider returning a conservative value (e.g., 0) instead of reverting, or otherwise ensure invariants make `obligations > gross` impossible in all intended operational states (including misreports/edge cases).

LOW

Fee-on-transfer / deflationary tokens break SmartAccountWrapper accounting (allocatedAssets overstatement), potentially causing stuck withdrawal requests

Location:

```
src/SmartAccountWrapper.sol:112-123
```

Description:

SmartAccountWrapper assumes the full `assets` amount is successfully transferred to the smart account and increments `allocatedAssets` by `assets`:

```
function _transferToSmartAccount(uint256 assets) internal {
  ...
  _getSmartAccountWrapperStorage().allocatedAssets += assets;
  IERC20(asset()).safeTransfer(_smartAccount, assets);
}
```

For fee-on-transfer / deflationary ERC20s, the smart account can receive **less** than `assets` even though the transfer succeeds.

This causes `allocatedAssets` to overstate the real assets held outside the vault, which then:

- Inflates `totalAssets()` (because `allocatedAssets()` is included)
- Misprices ERC4626 conversions (`previewRedeem`, `previewDeposit`, etc.)
- Can result in users creating withdrawal requests (burning shares) for asset amounts that can never become claimable, effectively **locking value** until manual intervention.

Additionally, if the token charges a fee on `transferFrom` into the wrapper, the subsequent transfer of `assets` out may revert due to insufficient balance, making deposits unexpectedly revert.

Impact:

Medium likelihood (depends on underlying token choice) with potentially high user impact:

- Mispricing and stuck async withdrawals.

- Unexpected deposit failures.

Recommendation:

Explicitly disallow fee-on-transfer/rebasing tokens (document + enforce), or account based on actual received/sent amounts by measuring balances before/after transfers and updating `allocatedAssets` with the delta.

LOW

MAX_DEVIATION_RATE safeguard is bypassable by resetting allocatedAssets to zero, allowing arbitrary jumps in reported allocatedAssets

Locations:

`src/SmartAccountWrapper.sol:143-151`

`src/SmartAccountWrapper.sol:153-167`

Description:

`SmartAccountWrapper.transmitAllocatedAssets()` attempts to limit how much the smartAccount can change the reported `allocatedAssets` via `_checkMaxDeviationRate()`.

However, `_checkMaxDeviationRate()` **skips validation entirely when current `allocatedAssets == 0`**:

```
uint256 _allocatedAssets = allocatedAssets();
if (_allocatedAssets > 0) {
  ... check deviation vs _allocatedAssets ...
}
```

(`src/SmartAccountWrapper.sol:143-151`)

Since the smartAccount can always call `transmitDeallocatedAssets(0)` (it only enforces `remainingAllocatedAssets <= current`), it can reset the stored value to 0 and then call `transmitAllocatedAssets(X)` with an arbitrary `X` (as long as `pendingWithdrawals() == 0`). The deviation check will not run on the second call because `_allocatedAssets` is now 0.

Impact:

The intended 0.25% drift bound does not reliably constrain reported `allocatedAssets`. A compromised or buggy smartAccount can still arbitrarily manipulate the `allocatedAssets` value used in ERC4626 pricing (`totalAssets()`), enabling the manipulation/dilution/DoS scenarios described in Finding #3.

Recommendation:

Do not special-case `_allocatedAssets == 0` as “no check”. If an initial bootstrap exception is needed, restrict it to a one-time initialization phase or validate against an independent on-chain source (e.g., actual token balance of the smartAccount) before allowing large jumps.

LOW

No rescue/sweep mechanism for non-underlying tokens accidentally sent to SmartAccountWrapper (funds can be stuck)

Location:

```
src/SmartAccountWrapper.sol:1-219
```

Description:

`SmartAccountWrapper` has no function to recover ERC20 tokens (other than the normal ERC-4626/7540 asset flows for the configured `asset()`) that are accidentally transferred to the wrapper contract.

Because there is also no `receive()`/`fallback()` withdraw path, any ETH sent via forced means (e.g., SELFDESTRUCT) is also unrecoverable.

Impact:

- Accidental transfers of unrelated ERC20s to the wrapper are permanently stuck.
- Forced ETH transfers are permanently stuck.

Recommendation:

Consider adding an owner-governed rescue/sweep function for non-`asset()` tokens (and optionally ETH), with clear constraints to prevent pulling the underlying `asset()` owed to share/request holders.

LOW

Owner can redirect and extract vault assets via setSmartAccount() + allocateAssets() (centralization / admin-drain risk)

Locations:

`src/SmartAccountWrapper.sol:169-180`

`src/SmartAccountWrapper.sol:118-123`

Description:

`SmartAccountWrapper` gives the `owner` unilateral power to: 1) change the `smartAccount` address (`setSmartAccount`), and 2) transfer idle vault assets to that `smartAccount` (`allocateAssets` ' `_transferToSmartAccount` ' `safeTransfer`).

```
function setSmartAccount(address smartAccount_) public onlyOwner {
 if (smartAccount_ == address(0)) revert SA__ZeroAddress();
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
}

function allocateAssets(uint256 assets) public onlyOwner {
 uint256 idle = idleAssets();
 if (assets > idle) revert SA__NotEnoughIdleAssets(assets, idle);
 _transferToSmartAccount(assets);
}

function _transferToSmartAccount(uint256 assets) internal {
 address _smartAccount = smartAccount();
 if (_smartAccount == address(0)) revert SA__SmartAccountNotSet();
 _getSmartAccountWrapperStorage().allocatedAssets += assets;
 IERC20(asset()).safeTransfer(_smartAccount, assets);
}
```

Impact:

If the owner key is compromised or malicious, the owner can:

- redirect the `smartAccount` to an attacker-controlled address, and
- transfer any currently-idle vault assets to that address.

Even if most assets are typically held in the smartAccount already, this is still a powerful admin-controlled funds movement primitive with no timelock/multisig enforcement at the contract level.

Recommendation:

If the system is meant to be non-custodial, remove or strongly constrain this power (e.g., immutable smartAccount, governance timelock, multisig-only owner, or invariant checks that the new smartAccount is an expected/proven account). Otherwise, explicitly document this as a trust assumption for users.

LOW

SmartAccountWrapper implements IERC1271 but does not advertise it via supportsInterface (ERC-165 detection mismatch)

Locations:

`src/SmartAccountWrapper.sol:16`

`src/SmartAccountWrapper.sol:204-218`

Description:

`SmartAccountWrapper` explicitly implements `IERC1271`:

```
contract SmartAccountWrapper is ... , SemiAsyncRedeemVault, IERC1271 {
```

but its `supportsInterface` implementation does not report support for the IERC1271 interface id (which is the selector of `isValidSignature(bytes32,bytes)` = `0x1626ba7e`).

Evidence:

```
function supportsInterface(bytes4 interfaceId) external pure returns (bool) {
 return interfaceId == type(IERC165).interfaceId
 || interfaceId == type(IERC7540Redeem).interfaceId
 || interfaceId == type(IERC7540Operator).interfaceId;
}
```

Impact:

Integrations that use ERC-165 detection to discover ERC-1271 support may incorrectly conclude the contract does not support ERC-1271, despite `isValidSignature` being implemented.

Recommendation:

Include `type(IERC1271).interfaceId` (or the `0x1626ba7e` interface id) in `supportsInterface`, or clearly document that ERC-165 detection is not provided for ERC-1271.

LOW

Upgradeable implementation lacks constructor `_disableInitializers()`, allowing implementation takeover

Locations:

`src/SmartAccountWrapper.sol:16-95`

`script/utils/DeployHelper.sol:27-43`

Description:

`SmartAccountWrapper` is an `Initializable` upgradeable implementation intended to be used behind a `BeaconProxy` (`SmartAccountProxy`). However, it has **no constructor** that calls `_disableInitializers()`. This leaves the **implementation contract itself** initializable by anyone after deployment.

Deploy scripts explicitly deploy the implementation as a standalone contract (`type(SmartAccountWrapper).creationCode`) before deploying the beacon, so the implementation address is live on-chain.

While initializing the implementation does not directly modify proxy state, it commonly leads to:

- **Implementation takeover/confusion**: an attacker can set themselves as `owner` on the implementation instance.
- **Theft of mistakenly sent tokens**: if assets are accidentally transferred to the implementation address (common operational mistake), the attacker-controlled implementation can be used to move/withdraw them (since it becomes a functional vault instance after initialization).
- Increased operational risk during incident response/monitoring, because the implementation appears “owned/initialized” by an attacker.

Evidence:

`SmartAccountWrapper` has an initializer but no constructor:

```
contract SmartAccountWrapper is Initializable, OwnableUpgradeable, SemiAsyncRedeemVault,
IERC1271 {
 ...
 function initialize(...) public initializer {
 __Ownable_init(owner_);
 __ERC20_init_unchained(name_, symbol_);
 __ERC4626_init_unchained(IERC20(underlyingToken_));
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
 }
}
```

Implementation is deployed directly in `DeployHelper.deployAll()`:

```
result.implementation = CREATE3.deployDeterministic(
 type(SmartAccountWrapper).creationCode,
 implSalt
);
```

Impact:

Low to Medium operational/security impact depending on operational practices:

- Direct protocol funds in proxies are not immediately at risk from this alone.
- Any tokens sent to the implementation address can be stolen.

Recommendation:

Add a constructor to the concrete implementation that calls `_disableInitializers()` (standard OZ pattern) so the implementation cannot be initialized directly.

LOW

Vault/wrapper accounting is incompatible with rebasing assets (allocatedAssets is fixed while balances can change), leading to mispriced shares and potential insolvency/DoS

Locations:

`src/SemiAsyncRedeemVault.sol:143-160`

`src/SmartAccountWrapper.sol:54-107`

`src/SmartAccountWrapper.sol:118-167`

Description:

Both `SemiAsyncRedeemVault` and `SmartAccountWrapper` implicitly assume the underlying ERC20 balance only changes via explicit transfers.

- `SemiAsyncRedeemVault.totalAssets()` and `idleAssets()` rely on `IERC20(asset()).balanceOf(address(this))`.
- `SmartAccountWrapper` additionally relies on a manually-maintained `allocatedAssets` counter to represent assets held externally by `smartAccount`.

With **rebasing tokens** (positive or negative rebases), balances can change without transfers:

- The vault's on-chain balance can change unexpectedly, changing `idleAssets()` and claimability in ways that do not correspond to any request lifecycle.
- The `smartAccount`'s balance can rebase, but `allocatedAssets` will not update automatically, desynchronizing `totalAssets()` from actual custody.

Impact:

- Share pricing (`previewDeposit`, `previewRedeem`, etc.) can become incorrect.
- The vault can become insolvent relative to its accounting (or appear solvent when it is not).
- In extreme cases, `totalAssets()` can revert (`SA__TotalAssetsUnderflow`) if obligations exceed the mis-accounted gross assets, DoSing ERC4626 views and any operations that depend on them.

Recommendation:

Explicitly restrict supported assets to non-rebasing ERC20s (document and/or enforce at initialization), or implement robust balance-delta based accounting that tolerates rebases (including reconciling `allocatedAssets` against actual smartAccount holdings).

LOW

claim() authorizes the receiver even when receiver != controller, allowing a third party to bypass the controller/operator access model

Location:

```
src/SemiAsyncRedeemVault.sol:407-417
```

Description:

`SemiAsyncRedeemVault.claim()` allows the **request receiver** to claim assets even if they are not the request `controller` and have not been approved as an operator by the controller.

While the claim always transfers assets to `request.receiver` (so this does not redirect funds), it weakens the intended ERC-7540 access-control model where the **controller** is the party empowered to operate the request (potentially for compliance, batching, accounting, etc.). If an inheriting implementation ever sets `receiver != controller` (possible via `_requestWithdraw`), the receiver can unilaterally trigger settlement.

Evidence:

```
// ERC-7540: restrict claim to receiver, controller, or operator of controller
address caller = _msgSender();
if (caller != request.receiver && !_isAuthorized(caller, request.controller)) {
 revert SA__NotAuthorized();
}
```

In the current repo's own request paths, `receiver == controller`, so this is latent. However, the base contract is `abstract` and exposes `_requestWithdraw(...)` as `internal virtual`, making `receiver != controller` a realistic future extension.

Impact:

- **Access-control bypass (latent):** a non-controller receiver can force claims, bypassing controller workflow/controls.
- Potential for unexpected operational behavior in integrations expecting controller-only claim.

Recommendation:

Restrict claims to `controller` (or controller-approved operators) only, unless there is a deliberate design requirement for receiver-triggered settlement—if so, document it explicitly and ensure downstream systems assume receiver can trigger claim.

LOW

forceTransmitDeallocatedAssets can INCREASE allocatedAssets (despite name), enabling arbitrary share-price inflation and DoS

Location:

```
src/SmartAccountWrapper.sol:135-141
```

Description:

`forceTransmitDeallocatedAssets()` is intended to report a *decrease* in `allocatedAssets` after assets are deallocated. However, it allows the owner to set `allocatedAssets` to a **higher** value than the current one whenever `pendingWithdrawals() == 0`:

```
function forceTransmitDeallocatedAssets(uint256 remainingAllocatedAssets) public onlyOwner {
 if (remainingAllocatedAssets > allocatedAssets() && pendingWithdrawals() > 0) {
 revert SA__PendingWithdrawals();
 }
 _getSmartAccountWrapperStorage().allocatedAssets = remainingAllocatedAssets;
}
```

If `pendingWithdrawals() == 0`, the `if` condition is false even when `remainingAllocatedAssets > allocatedAssets()`, so the owner can arbitrarily **increase** the reported `allocatedAssets`.

Because `allocatedAssets()` is included in `SemiAsyncRedeemVault.totalAssets()` (and therefore ERC4626 conversions/previews), this enables:

- Arbitrary **share price manipulation** (minting too few shares for deposits or showing inflated redemption previews)
- Potential **DoS** if owner sets a value that causes arithmetic overflow in `totalAssets()` or makes accounting inconsistent.

Impact:

Owner-only but high impact:

- A compromised/malicious owner can misprice shares and extract value from other users (if this wrapper is used by multiple depositors).
- Even without malice, a mistaken call can permanently break core view functions and block deposits/redemptions.

Recommendation:

Enforce monotonic decrease semantics (like `transmitDeallocatedAssets()` does):

- `require(remainingAllocatedAssets <= allocatedAssets())`
- If the intent is to block *any* updates during withdrawals, use a clear guard (e.g., check outstanding obligations, not just `pendingWithdrawals()`), and document the invariant.

LOW

initialize() does not validate underlyingToken_ (or smartAccount_) leading to bricked vault or stuck deposits

Locations:

`src/SmartAccountWrapper.sol:83-94`

`lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:96-101`

`lib/openzeppelin-contracts-upgradeable/lib/openzeppelin-contracts/contracts/utils/Address.sol:114-118`

Description:

`SmartAccountWrapper.initialize()` accepts user-supplied `underlyingToken_` and stores it via `__ERC4626_init_unchained(IERC20(underlyingToken_))` without validating that it is a nonzero ERC-20 contract.

If `underlyingToken_` is `address(0)` or any EOA/non-contract address, the vault becomes unusable:

- `ERC4626Upgradeable.asset()` will return the invalid address.
- Core functions (`deposit`, `withdraw`, `totalAssets`, etc.) will revert when they attempt to call `IERC20(asset())....`.
  - `SafeERC20`/`Address.verifyCallResultFromTarget` explicitly reverts when the target has `code.length == 0` and returns empty data.

`initialize()` also does not validate `smartAccount_`; initializing it to `address(0)` leaves the wrapper in a state where deposits/allocations revert until the owner later calls `setSmartAccount`.

Evidence:

`initialize()` stores the unvalidated addresses:

```
function initialize(
 address owner_,
 address smartAccount_,
 address underlyingToken_,
 string memory name_,
 string memory symbol_
) public initializer {
 __Ownable_init(owner_);
 __ERC20_init_unchained(name_, symbol_);
 __ERC4626_init_unchained(IERC20(underlyingToken_));
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
}
```

OZ ERC4626 initializer does not enforce contract/nonzero:

```
function __ERC4626_init_unchained(IERC20 asset_) internal onlyInitializing {
 ...
 $._asset = asset_;
}
```

But `SafeERC20` calls `Address.functionCall`, which reverts for empty-code targets:

```
if (returndata.length == 0 && target.code.length == 0) {
 revert AddressEmptyCode(target);
}
```

Impact:

- **Permanent DoS / bricked deployment** if `underlyingToken_` is misconfigured (zero/EOA).
- A misconfigured `smartAccount_ == address(0)` makes the vault non-functional for deposits/allocations until corrected (operational DoS).

Recommendation:

Add strict validation in `initialize()`:

- `underlyingToken_ != address(0)` and `underlyingToken_.code.length > 0`.
- Consider requiring `smartAccount_ != address(0)` (or document clearly that it may be set later).

LOW

renounceOwnership() can permanently disable critical maintenance functions (smartAccount rotation / accounting recovery), risking stuck funds

Locations:

`lib/openzeppelin-contracts-upgradeable/contracts/access/OwnableUpgradeable.sol:94-96`

`src/SmartAccountWrapper.sol:129-180`

Description:

`SmartAccountWrapper` inherits `OwnableUpgradeable`, which exposes `renounceOwnership()`:

```
function renounceOwnership() public virtual onlyOwner {
 _transferOwnership(address(0));
}
```

In this system, `onlyOwner` gates multiple critical operational/maintenance functions:

- `setSmartAccount()` (required to rotate/replace the smartAccount)
- `allocateAssets()` (moves idle assets into the smartAccount)
- `forceTransmitAllocatedAssets()` / `forceTransmitDeallocatedAssets()` (manual accounting recovery)

If ownership is renounced (intentionally or accidentally), these functions become permanently unusable.

Impact:

High-impact operational failure mode:

- If the `smartAccount` is compromised/lost/misconfigured after ownership is renounced, it cannot be rotated.
- If accounting becomes inconsistent (e.g., `allocatedAssets` drift), the owner cannot use the recovery functions.
- Users may be unable to exit in practice if the smartAccount cannot return assets and the owner can no longer intervene.

Recommendation:

If renouncing ownership is not intended for this vault, override `renounceOwnership()` to revert, or gate it behind a timelock/multisig procedure. At minimum, document this as a critical operational hazard for deployments.

LOW

requestRedeem can silently do nothing (no burn, no request) when previewRedeem rounds assets to 0

Location:

```
src/SemiAsyncRedeemVault.sol:308-342
```

Description:

`requestRedeem()` computes `assets = previewRedeem(shares)` (rounding **down**) and then calls `_processRequest(assets, shares, ...)`.

If `previewRedeem(shares)` rounds to **0 assets** (common for small `shares` when `totalAssets` is small relative to `totalSupply`), `_processRequest` will:

- compute `assetsToWithdraw = 0`, so it will not call `_withdraw`
- compute `assetsToRequest = 0`, so it will not call `_requestWithdraw`
- return `bytes32(0)` (`requestRedeem` returns `requestId = 0`

This is a **silent no-op**: the function returns as if the request was fully satisfied immediately, but no shares are burned and no async request exists.

This contradicts the function's own NatSpec ("shares are burned") and violates ERC-7540's expectation that `requestRedeem` either requests the shares or reverts.

Evidence:

```
function requestRedeem(uint256 shares, address controller, address owner)
 public
 virtual
 returns (uint256 requestId)
{
 uint256 assets = previewRedeem(shares); // can be 0 due to floor rounding
 bytes32 withdrawKey = _processRequest(assets, shares, controller, controller, owner);
 return uint256(withdrawKey); // returns 0 when _processRequest does nothing
}

function _processRequest(uint256 assets, uint256 shares, ...)
 internal
 returns (bytes32)
{
 uint256 assetsToWithdraw = Math.min(assets, maxWithdraw(owner));
 uint256 assetsToRequest = assets - assetsToWithdraw; // == 0

 uint256 sharesToRedeem = _convertToShares(assetsToWithdraw, Math.Rounding.Ceil); // == 0
 uint256 sharesToRequest = shares - sharesToRedeem; // == shares (>0)

 if (assetsToWithdraw > 0) { ... } // skipped
 if (assetsToRequest > 0 && sharesToRequest > 0) { ... } // skipped
 return bytes32(0);
}
```

Example:
If `totalSupply` is large and `totalAssets` is very small (or vice versa due to losses), then `previewRedeem(1)` can be 0. For example with OZ's ERC4626 virtual shares/assets:

- `previewRedeem(1) = floor(1 * (totalAssets+1) / (totalSupply+1))`

If `totalAssets=1` and `totalSupply=1000`, `previewRedeem(1) = floor(2/1001) = 0`.

Impact:

- Users (and integrators) can get a **false success** (requestId=0) while nothing happened.
- Dust shares may become non-requestable via the ERC-7540 path, breaking expected UX and ERC-7540 semantics.

Recommendation:
Ensure `requestRedeem(shares, ...)` always consumes the requested `shares` (burn/escrow) and either:

- creates an async request even if the implied `assets` rounds to 0, or
- reverts when `assets == 0` / when the request would become a no-op.

LOW

requestRedeem/requestWithdraw allow zero controller/receiver; immediate withdraw path can transfer assets to address(0) (burn)

Locations:

`src/SemiAsyncRedeemVault.sol:279-292`

`src/SemiAsyncRedeemVault.sol:308-342`

`lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:278-299`

Description:

`SemiAsyncRedeemVault.requestWithdraw()` and `requestRedeem()` accept user-supplied `receiver`/`controller` addresses with **no zero-address validation**.

In `_processRequest`, the contract may perform an **immediate** withdrawal of idle liquidity:

```
if (assetsToWithdraw > 0) _withdraw(_msgSender(), receiver, owner, assetsToWithdraw, sharesToRedeem);
```

OZ `ERC4626Upgradeable._withdraw` does not validate `receiver != address(0)` and will call `SafeERC20.safeTransfer(asset, receiver, assets)`.

As a result, providing `controller == address(0)` (in `requestRedeem`) or `receiver == address(0)` (in `requestWithdraw`) can cause the vault to transfer underlying assets to `address(0)` (burn) for tokens that do not revert on zero-address transfers.

Even when the async request portion would later revert for authorization reasons (e.g., `_requestWithdraw` checks controller/operator), the immediate-withdraw branch happens **before** that check and can already move funds.

Evidence:

```
function requestRedeem(uint256 shares, address controller, address owner) public returns
(uint256) {
 ...
 bytes32 withdrawKey = _processRequest(assets, shares, controller, controller, owner);
}

function _processRequest(uint256 assets, uint256 shares, address controller, address receiver,
address owner)
 internal returns (bytes32)
{
 ...
 if (assetsToWithdraw > 0) _withdraw(_msgSender(), receiver, owner, assetsToWithdraw,
sharesToRedeem);
 ...
}
```

OZ `_withdraw`:

```
_burn(owner, shares);
SafeERC20.safeTransfer(IERC20(asset()), receiver, assets);
```

Impact:

- Potential permanent loss of user funds if a caller (or buggy integration) passes `address(0)` as controller/receiver and the underlying token permits transfer-to-zero.
- Even if many ERC-20s revert on `to == address(0)`, relying on token behavior is fragile and can lead to unexpected behavior with nonstandard assets.

Recommendation:

Explicitly reject `controller == address(0)` / `receiver == address(0)` (and consider validating `owner != address(0)` defensively) before performing any burn/transfer logic.

LOW

setOperator() allows operator=address(0), creating ambiguous approvals and potential integration/accounting errors

Location:

`src/SemiAsyncRedeemVault.sol:123-128`

Description:

`setOperator(address op, bool approved)` accepts an unvalidated `op` address. Setting `op = address(0)` is allowed and will update storage + emit `OperatorSet`.

While this may not be directly exploitable, it is an input validation issue that can:

- Create confusing state for off-chain indexers and UIs (treating zero-address as a real operator).
- Cause edge-case logic errors in integrations that assume operators are always nonzero.

Evidence:

```
function setOperator(address op, bool approved) external returns (bool) {
 $.operators[_msgSender()][op] = approved;
 emit OperatorSet(_msgSender(), op, approved);
 return true;
}
```

Impact:

Low direct on-chain impact; primarily risk of integration bugs and confusing approvals.

Recommendation:

Reject `op == address(0)` (and optionally `op == msg.sender`) to keep operator sets well-formed.

LOW

setSmartAccount can strand all underlying assets and brick redemptions (no migration / safety checks)

Locations:

`src/SmartAccountWrapper.sol:112-123`

`src/SmartAccountWrapper.sol:176-180`

Description:

`SmartAccountWrapper` forwards deposited underlying to `smartAccount` and tracks it via `allocatedAssets`. The owner can later call `setSmartAccount()` to change the configured smart account address.

However, `setSmartAccount()` performs **no migration** of already-forwarded tokens and has **no safety checks** (e.g., `allocatedAssets == 0`, no pending requests, etc.). After switching:

- the old smart account can still custody the previously forwarded assets
- the wrapper will start forwarding new deposits to the new smart account
- withdrawals/claims depend on the configured smart account returning assets to the wrapper, but the new smart account may not control the old funds

This can permanently brick fulfillment of existing shares and pending redeem requests, even in the absence of malicious behavior (simple operational mistake).

Vulnerable Code:

Assets are forwarded to the currently configured smart account:

```
function _transferToSmartAccount(uint256 assets) internal {
 address _smartAccount = smartAccount();
 if (_smartAccount == address(0)) revert SA__SmartAccountNotSet();
 _getSmartAccountWrapperStorage().allocatedAssets += assets;
 IERC20(asset()).safeTransfer(_smartAccount, assets);
}
```

But the smart account can be changed at any time without reconciling custody:

```
function setSmartAccount(address smartAccount_) public onlyOwner {
 if (smartAccount_ == address(0)) revert SA__ZeroAddress();
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
 emit SmartAccountSet(smartAccount_);
}
```

Impact:

High-impact operational footgun:

- Existing vault assets can become inaccessible to the wrapper.
- Pending redeem/withdraw requests may never become claimable.
- Share price/accounting becomes inconsistent (allocatedAssets may refer to assets held by a different address).

Recommendation:

Add explicit safety checks around `setSmartAccount` (e.g., require `allocatedAssets == 0` and no outstanding obligations) or implement an explicit, auditable migration flow that moves custody and reconciles accounting before switching smart accounts.

LOW

setSmartAccount() is an unconfirmed privileged-role transfer that can redirect all future deposits and grant transmitter privileges to an arbitrary address

Locations:

`src/SmartAccountWrapper.sol:176-180`

`src/SmartAccountWrapper.sol:112-123`

`src/SmartAccountWrapper.sol:69-78`

Description:

`SmartAccountWrapper.setSmartAccount()` lets the `owner` replace the `smartAccount` address at any time. This address is both: 1) The **sink** that receives all deposited underlying assets (via `_deposit` ' `_transferToSmartAccount`), and 2) A **privileged role** that can call `onlySmartAccount` functions (`transmitAllocatedAssets` / `transmitDeallocatedAssets`) that affect vault accounting.

However, the role transfer is **single-step** with no recipient confirmation and no validation that the new address is a compatible smart account contract. A compromised/malicious owner (or simple operator mistake) can:

- Redirect all future deposits/allocations to an arbitrary address.
- Assign the `onlySmartAccount` privileges to an arbitrary address (EOA or unrelated contract).

This is an access-control/privilege-transfer risk: the protocol's security and custody assumptions hinge on the correctness of this address.

Evidence:

Role transfer is owner-only and immediate:

```
function setSmartAccount(address smartAccount_) public onlyOwner {
 if (smartAccount_ == address(0)) revert SA__ZeroAddress();
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
 emit SmartAccountSet(smartAccount_);
}
```

Deposits forward underlying to `smartAccount`:

```
function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal
override {
 super._deposit(caller, receiver, assets, shares);
 _transferToSmartAccount(assets);
}

function _transferToSmartAccount(uint256 assets) internal {
 address _smartAccount = smartAccount();
 ...
 IERC20(asset()).safeTransfer(_smartAccount, assets);
}
```

`smartAccount` controls privileged transmit functions:

```
modifier onlySmartAccount() { _checkSmartAccount(); _; }
```

Impact:

- **High-impact misconfiguration:** a wrong `smartAccount` address causes deposits to be sent to an unintended party, likely resulting in permanent loss.
- **Privilege reassignment without confirmation:** the new `smartAccount` can immediately exercise `onlySmartAccount` privileges.

Likelihood is medium because it requires owner action/compromise, but this is exactly the type of privileged-operation abuse the access-control review targets.

Recommendation:

Harden the `smartAccount` role transfer:

- Use a 2-step `proposeSmartAccount` / `acceptSmartAccount` flow (recipient confirmation).
- Optionally validate `smartAccount_` is a contract and supports an expected interface (e.g., ERC-165 check) if that is part of the intended trust model.
- Consider restricting changes when there are outstanding obligations, or providing an explicit migration procedure.

INFO

Beacon proxy architecture creates single-point-of-failure upgrade authority for all wrapper instances

Locations:

`src/SmartAccountProxy.sol:6-8`

`script/utils/DeployHelper.sol:35-43`

Description:

The system uses an OpenZeppelin `BeaconProxy` (`SmartAccountProxy`) pointing to an `UpgradeableBeacon`. This has two upgradeability properties that are easy to mis-assume: 1) The beacon address is **immutable per proxy** (cannot be changed after deployment). 2) **All proxies using the same beacon share upgrade fate**: the beacon owner can upgrade the implementation for every proxy at once.

In `DeployHelper`, the beacon owner is set directly to `params.owner` (potentially an EOA) without any timelock/multisig enforcement.

This is not a code-execution bug by itself, but it is a major upgradeability risk: compromise/misconfiguration of the beacon owner can lead to malicious upgrades of every wrapper proxy tied to that beacon with no per-instance escape hatch.

Evidence:

Proxy ties to a beacon at construction:

```
contract SmartAccountProxy is BeaconProxy {
 constructor(address beacon, bytes memory data) BeaconProxy(beacon, data) {}
}
```

Beacon ownership configured during deployment:

```
result.beacon = CREATE3.deployDeterministic(
 abi.encodePacked(
 type(UpgradeableBeacon).creationCode,
 abi.encode(result.implementation, params.owner)
 ),
 beaconSalt
);
```

Impact:

- If the beacon owner is compromised, all wrapper proxies under that beacon can be upgraded to malicious logic.
- If the beacon owner is lost, upgrades (including critical fixes) are impossible.

Recommendation:

Operational: use a timelock and/or multisig for beacon ownership; document that the beacon is immutable per proxy and that all proxies share upgrade control via the beacon.

INFO

Initializer uses `_unchained` OZ initializers and skips `__Nonces_init()`, reducing upgrade safety

Locations:

```
src/SmartAccountWrapper.sol:83-94
```

```
src/SemiAsyncRedeemVault.sol:49-51
```

Description:

`SmartAccountWrapper` is upgradeable and inherits multiple OpenZeppelin `*Upgradeable` base contracts via `SemiAsyncRedeemVault` (`ERC4626Upgradeable`, `NoncesUpgradeable`) and directly (`OwnableUpgradeable`).

However, `initialize()`:

- Calls `_unchained` initializers directly (`__ERC20_init_unchained`, `__ERC4626_init_unchained`) instead of the chained `__ERC20_init` / `__ERC4626_init`.
- Does **not** call `__Nonces_init()` even though `SemiAsyncRedeemVault` inherits `NoncesUpgradeable`.

Today, this does not break runtime behavior (OZ v5's `__Nonces_init` is empty, and `__ERC20_init`/`__ERC4626_init` only delegate to their unchained counterparts). But this pattern is **not upgrade-robust**:

- If a future OpenZeppelin version adds meaningful logic to the chained initializer (or new parent initializers), upgrading the implementation (via the beacon) would leave the proxy in a partially initialized state.
- Skipping `__Nonces_init()` is especially risky if `NoncesUpgradeable` later adds required initialization or invariants.

This is a classic "works now, breaks on upgrade" issue.

Evidence:

Initializer uses unchained initializers:

```
function initialize(...) public initializer {
 __Ownable_init(owner_);
 __ERC20_init_unchained(name_, symbol_);
 __ERC4626_init_unchained(IERC20(underlyingToken_));
 _getSmartAccountWrapperStorage().smartAccount = smartAccount_;
}
```

`SemiAsyncRedeemVault` inherits `NoncesUpgradeable`:

```
abstract contract SemiAsyncRedeemVault is Initializable, ERC4626Upgradeable, NoncesUpgrade-
able, ISemiAsyncRedeemVault {
 ...
}
```

Impact:

Informational upgrade-safety risk:

- Future implementation upgrades may behave incorrectly or brick functionality if new initializer logic is introduced upstream and not executed.

Recommendation:

Prefer calling the chained initializers (not `_unchained`) and include `__Nonces_init()` in the initialization sequence to preserve upgrade safety and align with OZ guidance.

INFO

Unchecked subtraction in _outstandingObligations can wrap if invariant breaks, cascading into incorrect accounting / totalAssets reverts

Location:

`src/SemiAsyncRedeemVault.sol:448-455`

Description:

`_outstandingObligations()` computes `requested - claimed` inside an `unchecked` block and only documents (via a comment) the invariant `requested >= claimed`.

If this invariant is ever violated (e.g., via upgrade/storage corruption, bad integration, or unexpected future changes), the subtraction will underflow and wrap to a huge `uint256`, which then feeds:

- `totalAssets()` (can revert with `SA__TotalAssetsUnderflow()` or return nonsensical results depending on other values)
- `idleAssets()` / `pendingWithdrawals()`
- any ERC-4626 conversion/preview logic that depends on `totalAssets()`

This is a numerical correctness footgun because a single invariant break can cascade into system-wide DoS/misaccounting.

Evidence:

```
function _outstandingObligations() internal view returns (uint256) {
 uint256 requested = cumulativeRequestedWithdrawalAssets();
 uint256 claimed = cumulativeClaimedAssets();
 // assert requested >= claimed
 unchecked {
 return requested - claimed;
 }
}
```

(`src/SemiAsyncRedeemVault.sol:448-455`)

Impact:

Low likelihood under current code paths (invariant appears intended to hold), but high blast radius if violated: wrapped obligations corrupt `totalAssets()` and downstream math.

Recommendation:

Enforce the invariant with an explicit check (`if (claimed > requested) revert`) or remove `unchecked` so an underflow reverts immediately at the point of failure.

Advisory to Clients

Cecuro highly recommends scheduling a follow-up security assessment within six months of this report or immediately after any significant modifications to the codebase, whichever occurs first. This practice is essential for preserving the project's security posture and identifying potential vulnerabilities that may arise from code changes or updates.

Disclaimer

The smart contracts submitted for analysis have been reviewed using Cecuro.ai's AI security analysis system, applying industry-standard vulnerability detection patterns and best practices at the time of this report. The findings disclosed in this report reflect the AI system's assessment of the submitted source code.

This report contains no statements or warranties on the identification of all vulnerabilities or the overall security of the code. Any security review methodology may not detect all potential issues, particularly novel attack vectors, complex business logic flaws, or economic exploits. The report covers the code submitted at the specified version and may not be relevant after any modifications.

Do not consider this report as a final and sufficient assessment regarding the security, utility, or bug-free status of the code. We recommend complementing this analysis with additional independent audits and a public bug bounty program to strengthen the security posture of your smart contracts.

Smart contracts are deployed and executed on blockchain platforms. The platform, its programming language, compiler, and other related software may contain vulnerabilities beyond the scope of code-level analysis. Cecuro.ai cannot guarantee the explicit security of the analyzed smart contracts.

This report does not constitute financial, legal, or investment advice. It is not a recommendation to buy, sell, or invest in any token or project, nor an endorsement of any kind. Users should conduct their own independent due diligence.

© Cecuro, Inc 2026. All rights reserved.

cecuro

Agentic Smart Contract Auditing