yAudit

March 2026

Prepared for
Rise

Audited by
fedebianu
spalen

RISEx RLP Strategies

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

RISEx RLP contracts contains two Yearn V3 TokenizedStrategy-based USDC strategies for the RISEx perpetual exchange: a market-making strategy and a liquidation strategy. The system relies on a RISEx adapter for collateral management, order placement, position queries, and liquidation takeovers, with shared epoch-based withdrawal protection and whitelisted deposits.

1.2 Audit Scope

This audit covers 4 smart contracts totaling approximately 652 lines of code across 4 days of review.

```
src/
├── RISExLiquidationStrategy.sol
├── RISExMarketMakingStrategy.sol
├── extensions/
│   ├── EpochWithdrawalProtection.sol
│   └── RISExStrategy.sol
└── interfaces/
    ├── IRISExLiquidationStrategy.sol
    ├── IRISExMarketMakingStrategy.sol
    ├── IYearnV3.sol
    └── RISExTypes.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	5
Low	6
Informational	1

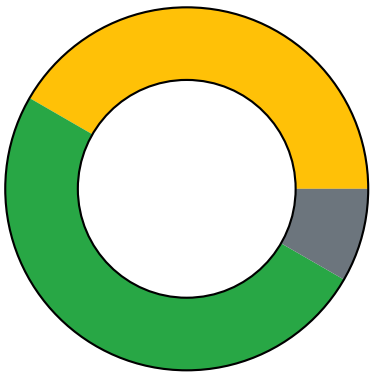

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The codebase is compact and generally readable, with clear separation between market-making, liquidation, and shared strategy logic. The main risk themes are stale accounting between reports and adapter-outage handling. The strongest issues identified concern business-critical fallback behavior and edge cases where users can interact against stale or incomplete state.

2 Audit Overview

2.1 Project Information

Protocol Name: Rise
Repository: https://github.com/risechain/risex-rlp-contracts
Commit Hash: 5d95ee88ff3caf477c7b66c82537f7366ed7ac5f
Commit URL: https://github.com/risechain/risex-rlp-contracts/commit/5d95ee88ff3caf477c7b66c82537f7366ed7ac5f

2.2 Audit Team

fedebianu, spalen

2.3 Audit Timeline

The audit was conducted from March 19 to 24, 2026.

2.4 Audit Resources

Code repositories

Category	Mark	Description
Access Control	Good	Role boundaries are mostly explicit and use Yearn's management, keeper, and emergency-admin model consistently, but privileged operational flows remain important to safe recovery.
Mathematics	Average	Core math is straightforward, but accounting around live equity, stale reports, and withdrawal-limit interactions creates subtle edge cases under loss or outage conditions.
Complexity	Average	The repository is relatively small, but the effective complexity is increased by the out-of-scope RISEx adapter dependency (which handles all fund movements and equity reads) and by the interaction between Yearn's TokenizedStrategy accounting, and epoch-based withdrawal throttles.
Libraries	Good	The code relies on established components such as Yearn TokenizedStrategy, OpenZeppelin, and Solady, which improves baseline reliability.
Decentralization	Low	The system is operationally centralized around management, keepers, whitelisted depositors, and emergency controls, so safe operation depends on privileged actors. This involves sending correct orders to the adapter and triggering reports when necessary.
Code Stability	Good	The contracts are compact and structurally stable, but several edge cases suggest the implementation is still iterating on outage handling and shutdown semantics.
Documentation	Good	README and inline comments provide a usable overview of intended strategy behavior, limits, and system architecture.
Monitoring	Average	The contracts emit useful events. Aside from events robust operational overall project monitoring is required for adapter health, triggering reports, and handling orders.
Testing and verification	Good	The project includes a solid Foundry suite with focused unit tests, regression coverage, and fork-based validation, which improves confidence in reproduced behaviors.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 Liquidation takeover caps can be overstated on equity-query failure

The liquidation strategy enforces takeover acceptance limits as percentages of total assets (TVL). When computing these limits in `_getLiveTotalAssets()`, the `catch` branch returns only `idle` when the equity query fails, ignoring negative equity. This fallback feeds into the state-changing takeover execution functions `executeIsolatedTakeover()` and `executeCrossTakeover()`, where it inflates daily caps, allows takeovers that should be blocked, writes exposure counters against the wrong TVL basis, and executes the actual position transfer on the adapter.

Technical Details

`_getLiveTotalAssets()` wraps the equity query in a try-catch:

```
function _getLiveTotalAssets() internal view returns (uint256) {
    uint256 idle = ERC20(asset).balanceOf(address(this));

    try i_adapter.getAccountEquity() returns (int256 equity18) {
        int256 equity = equity18 / int256(RISEX_DECIMALS_ADJUSTMENT);
        if (equity >= 0) {
            return idle + uint256(equity);
        }
        uint256 loss = uint256(-equity);
        return idle > loss ? idle - loss : 0;
    } catch {
        return idle; // ignores negative equity
    }
}
```

While this try-catch fallback is appropriate for view functions (`canAcceptTakeover`, `getMarketDailyCap`, `getGlobalDailyCap`), the same function is called from state-changing execution paths:

- `executeIsolatedTakeover()` calls `_getLiveTotalAssets()` on line 212
- `executeCrossTakeover()` calls it via `_previewCrossTakeover()` on line 504

In these paths, the inflated TVL basis flows into `_requireTakeoverWithinLimits()` which computes caps as `totalAssets * capBps / 10_000`. Positions that pass the inflated caps are then recorded via `_updateDailyExposure()` (state write) and executed via `i_adapter.executeIsolatedTakeover()` / `i_adapter.executeCrossTakeover()`.
Scenario:

```
RLP state: idle = 50k USDC, equity = -30k (positions underwater)
Real TVL = 50k - 30k = 20k

Oracle goes down -> getAccountEquity() reverts
_getLiveTotalAssets() catches -> returns idle = 50k (ignores -30k equity)

Real caps:                          Inflated caps:
  marketCap = 20k * 20% = 4k          marketCap = 50k * 20% = 10k
  globalCap = 20k * 50% = 10k         globalCap = 50k * 50% = 25k

A $9k takeover:
  Real:     9k > 4k market cap -> BLOCKED
  Inflated: 9k < 10k market cap -> PASSES, state written, takeover executed
```

PoC (extends `LiquidationSetup`):

```
contract PoC_TakeoverCapsInflatedOnOracleFailure is LiquidationSetup {
    function test_PoC_takeoverPassesWithInflatedCaps() public {
        // 1. Deposit 100k to RLP, harvest healthy
        depositToLiq(user, 100_000e6);
        mockLiqAccountEquity(int256(100_000e18));
        mockLiqWithdrawableAmount(100_000e18);
        reportLiq();

        // 2. Give strategy 50k idle (e.g. from prior freeCollateral)
        deal(address(asset), address(liqStrategy), 50_000e6);

        // 3. Equity oracle goes down — getAccountEquity reverts
        //    Real TVL = idle(50k) + equity(-30k) = 20k
        //    But _getLiveTotalAssets catches and returns idle(50k) only
        vm.mockCallRevert(
            liqAdapterAddr, abi.encodeWithSelector(IRISExAdapter.getAccountEquity.
selector), "ORACLE_DOWN"
        );

        // Other adapter functions still work (partial failure)
        address liquidatedAccount = address(0xBAD);
        int128 posSize = 0.3e18; // 0.3 BTC
        uint256 markPrice = 30_000e18; // $30k -> notional = 9k

        mockPosition(BTC_MARKET_ID, liquidatedAccount, posSize, -9_000e18, 0, MarginMode.
Isolated);
        mockMarkPrice(BTC_MARKET_ID, markPrice);
        mockIsolatedTakeover(liquidatedAccount, BTC_MARKET_ID);

        // 4. With real TVL (20k): marketCap = 4k, 9k > 4k -> should revert
        //    With inflated TVL (50k): marketCap = 10k, 9k < 10k -> passes
        uint256 realTVL = 20_000e6;
        uint256 realMarketCap = (realTVL * 2000) / 10_000; // 4k
        assertGt(9_000e6, realMarketCap, "9k > 4k real market cap - should be blocked");

        uint256 inflatedTVL = 50_000e6;
        uint256 inflatedMarketCap = (inflatedTVL * 2000) / 10_000; // 10k
        assertLt(9_000e6, inflatedMarketCap, "9k < 10k inflated market cap - passes");

        // 5. Takeover executes with inflated caps — state written, position transferred
        vm.prank(keeper);
        liqStrategy.executeIsolatedTakeover(liquidatedAccount, BTC_MARKET_ID);

        uint128 globalExposure = liqStrategy.getDailyGlobalExposure();
```

```
        assertEq(globalExposure, 9_000e6, "CONFIRMED: 9k takeover executed with inflated caps");
        // This takeover should NOT have been allowed with real TVL of 20k
        // The 9k notional exceeds the real market daily cap of 4k
    }
}
```

Impact

Medium. When the adapter's equity query fails but takeover operations remain functional, the strategy accepts liquidation positions whose notionals exceed the intended limits. Exposure counters are permanently written against the wrong TVL basis, and the actual position transfer on the adapter is irreversible. Depositors bear additional liquidation exposure beyond their configured risk budget during these windows. The impact is worse when equity is deeply negative — the larger the unreported loss, the bigger the gap between real caps and inflated caps.

Recommendation

Revert takeover operations when equity cannot be queried rather than proceeding with a potentially incorrect TVL basis. Remove the try/catch block if possible, or keep it only for view functions. Create a separate function for the state-changing path that does not use try/catch:

```
function _getLiveTotalAssetsOrRevert() internal view returns (uint256) {
    uint256 idle = ERC20(asset).balanceOf(address(this));
    int256 equity18 = i_adapter.getAccountEquity(); // reverts propagate
    int256 equity = equity18 / int256(RISEX_DECIMALS_ADJUSTMENT);
    if (equity >= 0) {
        return idle + uint256(equity);
    }
    uint256 loss = uint256(-equity);
    return idle > loss ? idle - loss : 0;
}
```

Developer Response

Fixed in commit 8033308

2.7.2 `RISExStrategy_tryGetWithdrawable()` fallback returns raw idle and bypasses epoch withdrawal throttling

Both strategies short-circuit to `return idle` when `getWithdrawableUSD()` reverts inside `_tryGetWithdrawable()`. This bypasses the epoch withdrawal limiter exactly on the path where idle-served withdrawals matter most, exposing more than the configured epoch budget during adapter read outages.

Technical Details

`RISExStrategy._tryGetWithdrawable()` wraps `i_adapter.getWithdrawableUSD()` in a `try/catch` and returns `(false, 0)` on any revert:

```
function _tryGetWithdrawable() internal view returns (bool, uint256) {
    try i_adapter.getWithdrawableUSD() returns (uint256 w) {
        return (true, w / RISEX_DECIMALS_ADJUSTMENT);
    } catch {
        return (false, 0);
    }
}
```

Both `RISExMarketMakingStrategy.availableWithdrawLimit()` and `RISExLiquidationStrategy.availableWithdrawLimit()` interpret that result as:

```
(bool registered, uint256 withdrawable) = _tryGetWithdrawable();
if (!registered) return idle;
```

This is inconsistent with the epoch throttling model in `EpochWithdrawalProtection._getEpochWithdrawalLimit()`. The epoch limiter is supposed to cap total withdrawals in the current epoch, including withdrawals served directly from idle. Returning raw `idle` on the fallback path skips `_getEpochWithdrawalLimit(idle)` entirely, so the strategy can expose the full loose balance even when the remaining epoch capacity is smaller.
Example:

- epoch-start assets = 100,000 USDC,
- `maxEpochOutflowBps = 20%`, so the remaining epoch budget is 20,000 USDC,
- the strategy holds 30,000 USDC idle,
- `getWithdrawableUSD()` reverts due to an adapter read outage.

On the healthy path, `availableWithdrawLimit()` is capped to 20,000 USDC. On the fallback path, it returns the full 30,000 USDC idle balance instead. A withdrawal served from idle then exceeds the configured epoch budget by 10,000 USDC.
The current PoC exercises the market-making strategy, but the liquidation strategy has the same raw-idle fallback in its withdrawal-limit hook.

PoC

Put this test in `test/strategy/RISExMarketMakingStrategy.t.sol`:

```
function test_poc_withdrawableQueryRevert_bypassesEpochThrottleForIdle() public {
    deposit(user, 100_000e6);
    report();

    deal(address(asset), address(strategy), 30_000e6);
    uint256 idle = asset.balanceOf(address(strategy));
    assertEq(idle, 30_000e6, "test requires idle funds above the remaining epoch capacity");

    uint256 epochLimit = strategy.getEpochWithdrawalCapacity(idle);
    assertEq(epochLimit, 20_000e6, "20 percent epoch cap should be lower than idle");

    mockWithdrawableAmount(0);
    uint256 capped = strategy.availableWithdrawLimit(user);
    assertEq(capped, epochLimit, "successful adapter reads should keep the epoch cap in force");

    clearMocks();
    vm.mockCallRevert(adapterAddr, abi.encodeWithSignature("getWithdrawableUSD()"), abi.encode("WITHDRAWABLE_REVERT"));

    uint256 bypassed = strategy.availableWithdrawLimit(user);
```

```
20      assertEq(bypassed, idle, "reverting withdrawable reads expose the full idle balance"
    );
21      assertGt(bypassed, epochLimit, "the returned limit exceeds the remaining epoch
    budget");

23      clearMocks();
24  }
```

Impact

Medium. This issue requires an adapter outage or partial read failure on `getWithdrawableUSD()`, but when it occurs it disables a core rate-limiting control for any funds already sitting idle. Users can withdraw more than the configured epoch budget, accelerating outflows exactly when adapter health is degraded.

Recommendation

Do not return raw `idle` on `_tryGetWithdrawable()` failure. If the adapter withdrawable query is unavailable, normal withdrawals should fail closed rather than continue on partial information. Applying `min(idle, _getEpochWithdrawalLimit(idle))` would fix the narrow epoch-throttle bypass, but it still allows users to exit against raw idle while adapter state is unavailable. If there are unknown or unreported losses during the outage window, early idle-served withdrawals can still receive an economic advantage over remaining holders.
The cleaner fix is to remove the `_tryGetWithdrawable()` fallback from normal withdrawal-limit paths and block ordinary withdrawals when `getWithdrawableUSD()` cannot be queried. If the protocol wants a recovery path during adapter outages, reserve that behavior for explicit shutdown mode: after `shutdownStrategy()`, management can use the existing `emergencyWithdraw()` flow to pull idle funds, and `availableWithdrawLimit()` may then return only the hard idle balance under shutdown-specific semantics.

Developer Response

Fixed in commit 316890f

2.7.3 Liquidation strategy lets users redeem at stale NAV after unreported losses

If the liquidation strategy experiences a temporary drawdown between a takeover and the next `report()`, withdrawals are still priced against the stale recorded NAV instead of live assets. An early-withdrawing user can therefore redeem above their fair live pro-rata claim, and the resulting deficit is shifted onto the remaining users.

Technical Details

`RISExStrategy._harvestAndReport()` records total assets as `idle + live equity` at report time and stores that snapshot in `TokenizedStrategy.totalAssets()`. Between reports, live equity can move while the recorded NAV remains stale.
During that stale-accounting window, `RISExLiquidationStrategy._freeFunds()` simply frees the requested amount subject to `withdrawable` and the epoch cap:

```
uint256 withdrawable = getWithdrawableAmount();
uint256 epochLimit = _getEpochWithdrawalLimit(idle);
uint256 maxAvailable = withdrawable < epochLimit ? withdrawable : epochLimit;
uint256 toWithdraw = amount > maxAvailable ? maxAvailable : amount;
```

`availableWithdrawLimit()` follows the same model and returns `idle + min(withdrawable, epochLimit)`. Neither path compares live assets against the stale recorded NAV, and neither caps a withdrawing owner's payout to their live pro-rata claim. This matters because withdrawals are initiated through `TokenizedStrategy._withdraw()`, which burns shares against the recorded accounting state and then asks the strategy to free the requested assets. If the liquidation book is temporarily underwater before the next report, the strategy still serves withdrawals at the stale NAV as long as `withdrawable` and the epoch cap allow it.
Example:

- recorded assets = 100,000 USDC,
- live assets = 80,000 USDC,
- user redeems shares worth 50,000 USDC at the stale snapshot.

The fair live payout is only `50,000 * 80,000 / 100,000 = 40,000 USDC`, but the current implementation can still pay the full `50,000 USDC`. The extra `10,000 USDC` is extracted by the first withdrawing user and socialized to the remaining users.
This state is realistic whenever an acquired position moves adversely, incurs funding, or is unwound at a worse mark before the next `report()`. The code does not enforce the assumption that liquidation inventory is always profitable between reports, so user withdrawal timing becomes economically relevant.
PoC

```
contract MockAdapterLiqStaleLossPoC {
    IERC20 internal immutable i_asset;

    int256 internal s_equity18;
    uint256 internal s_withdrawable18;

    constructor(address asset_) {
        i_asset = IERC20(asset_);
    }

    function setEquity(int256 equity18) external {
        s_equity18 = equity18;
    }

    function setWithdrawable(uint256 withdrawable18) external {
        s_withdrawable18 = withdrawable18;
    }

    function deployCollateral(address asset_, uint256 amount) external {
        require(asset_ == address(i_asset), "unexpected asset");
        require(i_asset.transferFrom(msg.sender, address(this), amount), "transferFrom
failed");

        s_equity18 += int256(amount * 1e12);
        s_withdrawable18 += amount * 1e12;
    }

    function freeCollateral(address asset_, uint256 amount) external returns (uint256
freed) {
        require(asset_ == address(i_asset), "unexpected asset");
```

```
29          require(i_asset.transfer(msg.sender, amount), "transfer failed");

31          int256 amount18 = int256(amount * 1e12);
32          s_equity18 -= amount18;

34          uint256 scaledAmount = amount * 1e12;
35          s_withdrawable18 = scaledAmount > s_withdrawable18 ? 0 : s_withdrawable18 -
    scaledAmount;
36          return amount;
37      }

39      function getAccountEquity() external view returns (int256) {
40          return s_equity18;
41      }

43      function getWithdrawableUSD() external view returns (uint256) {
44          return s_withdrawable18;
45      }

47      function getCollateralBalance(address asset_) external view returns (int256) {
48          require(asset_ == address(i_asset), "unexpected asset");
49          return s_equity18;
50      }
51  }

53  function test_poc_liq_withdraw_overpaysDuringUnreportedLoss() public {
54      MockAdapterLiqStaleLossPoC mockAdapter = new MockAdapterLiqStaleLossPoC(address(
    asset));
55      RISExLiquidationStrategy localStrategy =
56          new RISExLiquidationStrategy(address(asset), "RISEx Liquidation PoC", address(
    mockAdapter));
57      IStrategy localIStrategy = IStrategy(address(localStrategy));

59      localStrategy.setWhitelistedDepositor(user, true);
60      localStrategy.setMaxEpochOutflowBps(10_000);

62      uint256 depositAmount = 100_000e6;
63      deal(address(asset), user, depositAmount);

65      vm.startPrank(user);
66      asset.approve(address(localStrategy), depositAmount);
67      localIStrategy.deposit(depositAmount, user);
68      vm.stopPrank();

70      // Crystallize a 100k accounting snapshot before introducing the stale loss.
71      localIStrategy.report();

73      // Live value falls to 80k, but recorded totalAssets remains 100k until the next
    report.
74      deal(address(asset), address(mockAdapter), 80_000e6);
75      mockAdapter.setEquity(80_000e18);
76      mockAdapter.setWithdrawable(80_000e18);

78      uint256 requestedAssets = 50_000e6;
79      uint256 fairPayout = (requestedAssets * 80_000e6) / depositAmount;
80      uint256 sharesToRedeem = localIStrategy.convertToShares(requestedAssets);

82      uint256 balanceBefore = asset.balanceOf(user);
83      vm.prank(user);
84      localIStrategy.redeem(sharesToRedeem, user, user);
85      uint256 received = asset.balanceOf(user) - balanceBefore;

87      assertEq(received, requestedAssets, "withdrawer exits at stale recorded value");
```

```
88      assertEq(fairPayout, 40_000e6, "fair live payout should be 40k");
89      assertGt(received, fairPayout, "withdrawer is overpaid relative to live assets");
90      assertEq(asset.balanceOf(address(mockAdapter)), 30_000e6, "remaining holders keep
    only 30k backing");
91  }
```

Impact

Medium. During temporary unreported drawdowns, a first-withdrawing user can redeem more USDC than their fair live pro-rata claim. The excess is shifted onto the remaining users, who continue to appear whole until the next report crystallizes the loss. The extraction per window is bounded by the epoch throttle, but the loss shifting is still economically meaningful and depends only on withdrawal timing rather than merit.

Recommendation

Do not serve normal withdrawals against a stale recorded NAV when live assets are below recorded assets.
The safest fix is to make both `availableWithdrawLimit()` and `_freeFunds()` block or return zero when `liveAssets < recordedAssets`, and require a `report()` before further withdrawals:

```
(uint256 idle, int256 equity, uint256 liveAssets, uint256 recordedAssets) =
_getLiveAssetState();

if (recordedAssets > 0 && liveAssets < recordedAssets) {
    return 0; // or revert in _freeFunds()
}
```

If the team wants to allow withdrawals during stale-loss windows, the fix must be applied consistently across both the limit calculation and the fund-freeing path using a live-value model. In particular, `availableWithdrawLimit(owner)` should not exceed the owner's live pro-rata claim `(liveAssets * balanceOf(owner)) / totalSupply()`, and `_freeFunds()` must honor the same live cap. Fixing only `_freeFunds()` is insufficient because idle-served withdrawals can bypass it.

Developer Response

Fixed in commit a4ec448

2.7.4 `RISExStrategy._freeFunds()` can cause over-withdrawal when idle assets coexist with unreported losses

When losses are stale and the strategy also holds local idle funds, the withdrawal flow applies the haircut only to the RISEx shortfall while still paying idle at face value. This lets an early withdrawer receive more than their fair share and leaves the remaining holders with a larger residual loss.

Technical Details

`TokenizedStrategy._withdraw()` first consumes any local idle balance and only calls `freeFunds()` for the shortfall `assets - idle`.
`RISExMarketMakingStrategy._freeFunds()` then computes a live-vs-recorded haircut using:

- `estimatedAssets = idle + equity` (or `idle - loss` when equity is negative),
- `recordedAssets = TokenizedStrategy.totalAssets()`,
- `toWithdraw = _amount * estimatedAssets / recordedAssets` when `estimatedAssets < recordedAssets`.

This is internally inconsistent because `_amount` is already net of idle, while `estimatedAssets` still includes that same idle balance. The strategy therefore applies the loss haircut only to the RISEx shortfall, but still lets the withdrawer receive the full idle amount on top.
Additionally, the Yearn TokenizedStrategy template explicitly warns against this pattern:

> *"Should not rely on asset.balanceOf(address(this)) calls other than for diff accounting purposes."*

The strategy uses `balanceOf` for absolute value computation in the pro-rata ratio, violating this guidance.
Example:

- recorded assets = 100,
- live assets = 80, made of `idle = 20` and `equity = 60`,
- user requests a 50-asset withdrawal.

`TokenizedStrategy` requests `RISExMarketMakingStrategy` to free only `30` (`50 - 20`). `_freeFunds()` scales that shortfall to `24` (`30 * 80 / 100`). The user then receives `20 + 24 = 44`, even though the fair pro-rata payout for a 50-asset withdrawal against an 80/100 live-to-recorded ratio is only `40`.
The extra value is socialized to the remaining shareholders, who are left with a larger share of the still-unreported loss. This state is reachable whenever the strategy has non-zero idle between reports, including after manual emergency withdrawals or any flow that leaves loose USDC in the strategy. The existing test suite already models non-zero idle balances as a normal state in withdrawal-limit calculations.
PoC

```
contract MockAdapterFreeFundsPoC {
    ERC20 internal immutable i_asset;
    int256 internal s_equity18;
    uint256 internal s_withdrawable18;

    constructor(address asset_) {
        i_asset = ERC20(asset_);
    }

    function setEquity(int256 equity18) external {
        s_equity18 = equity18;
    }
```

```
14      function setWithdrawable(uint256 withdrawable18) external {
15          s_withdrawable18 = withdrawable18;
16      }
17
18      function deployCollateral(address asset_, uint256 amount) external {
19          require(asset_ == address(i_asset), "unexpected asset");
20          i_asset.transferFrom(msg.sender, address(this), amount);
21
22          s_equity18 += int256(amount * 1e12);
23          s_withdrawable18 += amount * 1e12;
24      }
25
26      function freeCollateral(address asset_, uint256 amount) external returns (uint256
    freed) {
27          require(asset_ == address(i_asset), "unexpected asset");
28          i_asset.transfer(msg.sender, amount);
29
30          int256 amount18 = int256(amount * 1e12);
31          s_equity18 -= amount18;
32
33          uint256 scaledAmount = amount * 1e12;
34          s_withdrawable18 = scaledAmount > s_withdrawable18 ? 0 : s_withdrawable18 -
    scaledAmount;
35          return amount;
36      }
37
38      function getAccountEquity() external view returns (int256) {
39          return s_equity18;
40      }
41
42      function getWithdrawableUSD() external view returns (uint256) {
43          return s_withdrawable18;
44      }
45
46      function getCollateralBalance(address asset_) external view returns (int256) {
47          require(asset_ == address(i_asset), "unexpected asset");
48          return s_equity18;
49      }
50  }
51
52  function test_poc_freeFunds_overpaysWhenIdleCoexistsWithUnreportedLoss() public {
53      MockAdapterFreeFundsPoC mockAdapter = new MockAdapterFreeFundsPoC(address(asset));
54      RISExMarketMakingStrategy localStrategy =
55          new RISExMarketMakingStrategy(address(asset), "RISEx Strategy PoC", address(
    mockAdapter));
56      IStrategy localIStrategy = IStrategy(address(localStrategy));
57
58      localStrategy.setWhitelistedDepositor(user, true);
59      localStrategy.setMaxEpochOutflowBps(10_000);
60
61      uint256 depositAmount = 100_000e6;
62      deal(address(asset), user, depositAmount);
63      vm.startPrank(user);
64      asset.approve(address(localStrategy), depositAmount);
65      localIStrategy.deposit(depositAmount, user);
66      vm.stopPrank();
67
68      localIStrategy.report();
69
70      // Stale-loss state: 80k live assets split between 20k idle and 60k still on RISEx.
71      deal(address(asset), address(localStrategy), 20_000e6);
72      deal(address(asset), address(mockAdapter), 60_000e6);
73      mockAdapter.setEquity(60_000e18);
```

```
74      mockAdapter.setWithdrawable(60_000e18);
76      uint256 requestedAssets = 50_000e6;
77      uint256 fairPayout = (requestedAssets * 80_000e6) / depositAmount;
78      uint256 sharesToRedeem = localIStrategy.convertToShares(requestedAssets);
80      uint256 balanceBefore = asset.balanceOf(user);
81      vm.prank(user);
82      localIStrategy.redeem(sharesToRedeem, user, user);
83      uint256 received = asset.balanceOf(user) - balanceBefore;
85      assertEq(received, 44_000e6, "Withdrawer should receive 20k idle plus 24k from RISEx
    ");
86      assertEq(fairPayout, 40_000e6, "Fair payout at an 80/100 live ratio should be 40k");
87      assertGt(received, fairPayout, "Withdrawer is overpaid relative to live assets");
88      assertEq(asset.balanceOf(address(localStrategy)), 0, "Idle should be fully consumed"
    );
89      assertEq(asset.balanceOf(address(mockAdapter)), 36_000e6, "Remaining holders are
    left with only 36k backing");
90  }
```

Impact

Medium. Under unreported loss conditions, a withdrawer can receive more than their fair economic share if the strategy is holding local idle funds. The excess comes directly at the expense of the remaining shareholders, who are forced to absorb a larger portion of the stale loss on subsequent withdrawals or on the next report.

Recommendation

Under the current Yearn withdrawal flow, `TokenizedStrategy._withdraw()` consumes local idle before calling `_freeFunds()`, so `_freeFunds()` only receives the residual shortfall and cannot safely apply a fair whole-withdrawal haircut by itself. If the team does not want to change the shared Yearn withdrawal path, the practical fix is to fail closed during stale-loss windows rather than trying to continue withdrawals on partial information.
Specifically, the market-making strategy should treat `liveAssets < recordedAssets` as a temporary stale-loss state. In that state, normal withdrawals should be blocked until a fresh `report()` realizes the loss and refreshes accounting. `availableWithdrawLimit()` should therefore return zero when live assets are below recorded assets, and `_freeFunds()` should enforce the same rule defensively so execution cannot proceed on stale accounting even if a caller bypasses the view path.
If the team wants to preserve an emergency recovery path, once the strategy is shut down, users should withdraw only already recovered idle cash, even if stale losses still exist.
Another possibility is to implement `tend()` to deploy idle funds and `tendTrigger()` so the keeper can detect idle balances. That way, the strategy should not retain idle funds if the keeper is operating as expected.

Developer Response

Fixed in commit a4ec448

2.7.5 Negative equity harvest blocks strategy deposits

When `_harvestAndReport` reports `totalAssets = 0` (negative equity exceeding idle balance), TokenizedStrategy blocks all subsequent deposits with a `ZERO_SHARES` revert. Since the Yearn V3 Vault uses `strategy.deposit()` via `update_debt` to allocate funds to strategies, this means the vault **cannot push new debt** to the affected strategy. This could break the 95/5 allocation ratio and leave user funds sitting idle in the vault, undeployed and earning nothing.

Technical Details

In `RISExStrategy._harvestAndReport()`, when equity is negative and exceeds idle:

```
function _harvestAndReport() internal override returns (uint256 totalAssets_) {
    uint256 idle = ERC20(asset).balanceOf(address(this));
    int256 equity = getAccountEquity();

    if (equity >= 0) {
        totalAssets_ = idle + uint256(equity);
    } else {
        uint256 loss = uint256(-equity);
        totalAssets_ = idle > loss ? idle - loss : 0;
    }

    _resetEpoch(totalAssets_.toUint128(), idle.toUint128());
}
```

When `totalAssets = 0` but `totalSupply > 0` (existing shares), any call to `strategy.deposit()` would compute zero shares for the depositor, and TokenizedStrategy reverts with `ZERO_SHARES` to protect the caller.
The cascade:

1. Strategy equity goes negative (positions underwater) → harvest reports `totalAssets = 0`
2. `vault.update_debt(strategy, target, 0)` calls `strategy.deposit()` → reverts `ZERO_SHARES`
3. Vault cannot allocate debt to this strategy → funds sit as vault idle
4. The 95/5 ratio between MM and RLP strategies cannot be maintained
5. If MM is the blocked strategy, 95% of new deposits remain idle
6. If RLP is blocked, liquidation capacity shrinks (caps are TVL-based)

PoC

```
contract PoC_NegativeEquityBlocksDeposits is RISExSetup {
    address public vaultDepositor = address(20);

    function setUp() public override {
        super.setUp();
        whitelistDepositor(vaultDepositor);
    }

    /// @notice After harvest with totalAssets=0, strategy blocks ALL deposits (
including vault's update_debt)
    function test_PoC_negativeEquity_strategyBlocksDepositsAfterHarvest() public {
        // 1. User deposits 100k, harvest healthy
```

```
        deposit(user, 100_000e6);
        mockAccountEquity(int256(100_000e18));
        mockWithdrawableAmount(100_000e18);
        mockFreeCollateral(100_000e6);
        report();

        assertGt(iStrategy.totalAssets(), 0, "precondition: totalAssets > 0");

        // 2. Equity goes deeply negative: -120k, idle = 0
        //    loss(120k) > idle(0) → totalAssets = 0
        mockAccountEquity(int256(-120_000e18));
        mockWithdrawableAmount(0);

        // 3. Harvest reports totalAssets = 0
        report();
        assertEq(iStrategy.totalAssets(), 0, "totalAssets is 0 after negative equity
harvest");
        assertGt(iStrategy.totalSupply(), 0, "shares still exist");

        // 4. Strategy blocks deposits — ZERO_SHARES revert
        //    In production this is vault calling strategy.deposit() via update_debt
        //    Same code path, same revert
        deal(address(asset), vaultDepositor, 50_000e6);
        vm.startPrank(vaultDepositor);
        asset.approve(address(strategy), 50_000e6);
        vm.expectRevert("ZERO_SHARES");
        iStrategy.deposit(50_000e6, vaultDepositor);
        vm.stopPrank();

        // CONFIRMED: strategy is frozen for new deposits until equity recovers +
harvest
    }
}
```

Impact

Medium.

- **Vault debt allocation blocked**: `update_debt` to the affected strategy reverts, breaking the 95/5 ratio
- **Capital inefficiency**: User deposits into the vault succeed but funds sit idle, earning nothing
- **RLP liquidation capacity**: If RLP is blocked, the protocol cannot take over liquidated positions (daily caps scale with TVL which is now 0)
- **Recovery requires two steps**: equity must go positive AND a harvest must run — if the adapter is also down, recovery is impossible
- **Pre-harvest gap**: Between crash and harvest, deposits are accepted into a bankrupt account (stale totalAssets)

User deposits into the VaultV3 itself are not blocked — only the vault's ability to allocate those funds to strategies.

Recommendation

Remove the bankrupt strategy from the vault and replace it with a new strategy of the same type. This can be done in a single transaction: call `report()` with zero assets, remove the bankrupt strategy, and add the replacement strategy to the end of the vault queue.

Developer Response

Acknowledged.

2.8 Low Findings

2.8.1 `transferToMM()` signatures are not bound to the MM adapter destination

Technical Details

`RISExLiquidationStrategy.transferToMM()` verifies an EIP-712 signature over `(marketId, size, price, deadline, nonce)` using `TRANSFER_TO_MM_TYPEHASH`:

```
bytes32 public constant TRANSFER_TO_MM_TYPEHASH =
    keccak256(
        "TransferToMM(uint16 marketId,uint128 size,uint128 price,uint256 deadline,
uint256 nonce)"
    );
```

After signature verification, the function reads the mutable storage variable `s_mmAdapter` and uses it as the transfer destination when constructing the OTC request:

```
IRISExAdapter.OtcRequest memory request = IRISExAdapter.OtcRequest({
    marketId: marketId,
    mmVault: s_mmAdapter,
    side: ourSide,
    size: size,
    price: price,
    flags: 0,
    extraData: ""
});
```

The signed payload does not include the MM adapter address, and `setMMAdapter()` allows management to change `s_mmAdapter` after a signature is created but before it is executed. As a result, a valid outstanding signature can be executed against a different OTC destination than the signer reviewed.
This is a real authorization gap in the signature flow. The signer configured in `RISExStrategy` is distinct from `management`, and the EIP-712 path is clearly intended to bind execution to off-chain approved parameters. Today it binds price, size, market, deadline, and nonce, but not the destination vault/account.

Impact

Low. A keeper can execute a previously valid `transferToMM()` signature against a different MM adapter than the signer intended if `s_mmAdapter` changes before execution. This does not let an arbitrary caller forge transfers, but it weakens the guarantees of the signature-based authorization flow and can route inventory to an unintended vault/account configuration.

Recommendation

Include the MM adapter destination in the signed data and in the function parameters used to reconstruct the EIP-712 digest. At execution time, verify that the signed destination matches the current configured `s_mmAdapter` before forwarding the OTC request.
That can be done by extending the typehash to include `address mmAdapter`, passing `mmAdapter` into `transferToMM()`, and rejecting execution if the signed value and current storage value differ.

Developer Response

Acknowledged.

2.8.2 Deposit cap can fail open to stale reported assets when live equity queries revert

Technical Details

The `availableDepositLimit()` function in `RISExMarketMakingStrategy.sol:115-123` enforces the configured deposit cap by querying `_getLiveTotalAssetsForDepositCap()`. That helper is designed to compute the strategy's current total assets using live adapter equity when available. However, when `i_adapter.getAccountEquity()` reverts, `_getLiveTotalAssetsForDepositCap()` catches the failure and returns `TokenizedStrategy.totalAssets()` instead.
`TokenizedStrategy.totalAssets()` reflects the last reported assets value, which is synchronized only during `_harvestAndReport()` in `RISExStrategy.sol:86-99`. Between harvests, the strategy's actual live equity can change due to unreported profit or loss from trading activity. If live assets have grown beyond the reported value and the equity query fails during a deposit-limit check, `availableDepositLimit()` will compute remaining capacity using the stale lower reported assets rather than the current higher live assets.
This means the cap enforcement computes `max(0, depositCap - last reported total assets)` instead of the intended `max(0, depositCap - current live total assets)`. If the difference is material, a whitelisted depositor can deposit above the intended cap before the next harvest updates accounting.
The fail-open fallback path in `_getLiveTotalAssetsForDepositCap()` at line 224 allows deposits to proceed with stale data when the adapter view call fails. The code includes an explicit `try/catch` block around `i_adapter.getAccountEquity()`, indicating that query failures are anticipated, but there is no documentation showing that bypassing the cap under those conditions is intentional.

Impact

Low. A whitelisted depositor can deposit above `s_depositCap` when the strategy's live assets exceed reported assets and the adapter equity query reverts during the cap check. This weakens the deposit cap as a risk control, allowing more capital into the strategy than management intended. Excess total value locked can increase exposure to RISEx trading losses and undermine operational assumptions that the cap is binding. The issue does not directly drain funds or lock the system, but it materially weakens cap enforcement under specific failure conditions.

Recommendation

When `s_depositCap` is non-zero, fail closed if live equity cannot be queried. Do not substitute stale or synthetic asset values in the deposit-cap path. Instead, remove the `try/catch` fallback in `_getLiveTotalAssetsForDepositCap()` and let `availableDepositLimit()` revert when `i_adapter.getAccountEquity()` is unavailable, preventing deposits until adapter health is restored.

Developer Response

Fixed in commit 316890f

2.8.3 Epoch protection can lock first depositors until the first report

Both strategies initialize epoch protection with `epochStartAssets = 0`. Until the first successful `report()` seeds the epoch baseline, the withdrawal-limit hook can return zero even immediately after a user deposit. This is a real bootstrap quirk, but its operational impact is temporary and management-controlled.

Technical Details

`RISExStrategy` calls `_initEpochProtection(0)` in its constructor. That means `s_epochStartAssets` starts at zero and stays there until `_harvestAndReport()` eventually calls `_resetEpoch()`.
The epoch limiter in `EpochWithdrawalProtection._getEpochWithdrawalLimit()` derives its allowance directly from `s_epochStartAssets`:

```
uint256 maxWithdrawable = (s_epochStartAssets * s_maxEpochOutflowBps) /
EPOCH_BPS_DENOMINATOR;
```

If `s_epochStartAssets == 0`, then `maxWithdrawable == 0`, so the remaining epoch capacity is also zero.
Both concrete strategies use that helper in `RISExMarketMakingStrategy.availableWithdrawLimit()` and `RISExLiquidationStrategy.availableWithdrawLimit()`. As a result, a first depositor can enter a state where:

- funds have already been deposited,
- the strategy has not yet been reported,
- `availableWithdrawLimit()` still returns `0`,
- and normal user withdrawals remain blocked until management executes the first `report()`.

This is a bootstrap-only issue rather than a steady-state accounting bug, but it is still a real mismatch between "freshly deposited funds" and "withdrawable capacity" before the first report cycle has run.
PoC
Put this test in `test/strategy/RISExMarketMakingStrategy.t.sol`:

```
function test_poc_firstDeposit_isLockedUntilInitialReport() public {
    uint256 depositAmount = 100_000e6;
    deposit(user, depositAmount);

    assertEq(strategy.epochStartAssets(), 0, "constructor leaves epoch start assets at zero before first report");

    uint256 availableBeforeReport = strategy.availableWithdrawLimit(user);
    assertEq(availableBeforeReport, 0, "fresh deposits are locked until the first report resets epoch state");

    vm.expectRevert();
    vm.prank(user);
    iStrategy.withdraw(1e6, user, user);

    report();

    uint256 availableAfterReport = strategy.availableWithdrawLimit(user);
    assertGt(availableAfterReport, 0, "report should unblock withdrawals by setting epoch start assets");
}
```

Impact

Low. Deposits made before the first report can be temporarily non-withdrawable even though the funds have already entered the strategy. This does not create a lasting freeze or loss of funds, and the condition clears as soon as management executes the first `report()`. The issue is therefore best understood as a bootstrap liveness quirk, not a steady-state insolvency or drain vector.

Recommendation

Seed epoch state on the first funded balance, or bypass epoch throttling until the first successful report initializes `s_epochStartAssets`. If the current behavior is intentional, document that deposits are not expected before the initial report/bootstrap step.

Developer Response

Acknowledged. This is a bootstrap-only quirk: `epochStartAssets` is zero until the first `report()`, which management calls immediately after the first deposit as part of the deployment sequence.

2.8.4 Epoch withdrawal accounting can misclassify idle redeployment as user outflow and temporarily block withdrawals

If an epoch starts with non-zero idle, a later deposit will deploy the full loose balance, including that pre-existing idle, into RISEx. The epoch tracker then interprets the idle drop as if users had already withdrawn it, which can prematurely consume the epoch withdrawal budget and temporarily clamp withdrawals even though adapter collateral remains healthy.

Technical Details

`EpochWithdrawalProtection._getEstimatedIdleWithdrawals()` estimates idle-served withdrawals as:

```
if (s_epochStartIdle > currentIdle) {
    return s_epochStartIdle - currentIdle;
}
return 0;
```

That estimate is then charged against the current epoch budget in `_getEpochWithdrawalLimit()`.
The narrow problem is that a later deposit can reduce `currentIdle` without any user withdrawal having happened. `TokenizedStrategy._deposit()` transfers in the new assets and then calls:

```
IBaseStrategy(address(this)).deployFunds(
    _asset.balanceOf(address(this))
);
```

So the strategy deploys the entire loose balance currently held, not only the new deposit. If an epoch began with `s_epochStartIdle > 0`, a subsequent deposit can sweep that pre-existing idle into RISEx and reduce `currentIdle` toward zero. The epoch tracker then treats `s_epochStartIdle - currentIdle` as idle-served withdrawals even though the balance drop came from internal redeployment.
Example:

- a report starts a new epoch with `epochStartAssets = 100,000` and `epochStartIdle = 20,000`,
- `maxEpochOutflowBps = 20%`, so the epoch budget is `20,000`,
- no one withdraws,
- a later deposit of `1` USDC triggers `deployFunds(balanceOf(this))`, which also deploys the pre-existing `20,000` idle,
- `currentIdle` falls from `20,000` to `0`.

`_getEstimatedIdleWithdrawals(0)` now returns `20,000`, so `_getEpochWithdrawalLimit(0)` returns `0` even though no user outflow has occurred. Any strategy that uses this helper in `availableWithdrawLimit()` can then clamp withdrawals until the next report refreshes the epoch state.
This is much narrower than a generic “idle redeployment during the epoch” claim. In the current codebase, the realistic trigger is a post-report deposit sweeping pre-existing idle, because deposits deploy the full loose balance by design.

Impact

Low. This is a temporary liveness issue with narrow preconditions: the epoch must start with non-zero idle, and a later deposit must redeploy that idle before the next report. When it happens, withdrawals can be blocked or reduced even though collateral remains withdrawable on RISEx. Funds are not lost, and the condition clears on the next report or epoch reset.

Recommendation

Do not infer all decreases in `currentIdle` as user outflow. Track internal redeployments separately and exclude them from `_getEstimatedIdleWithdrawals()`, or maintain an adjusted idle baseline that is updated whenever `_deployFunds()` sweeps pre-existing loose balance. If that complexity is undesirable, document this as an explicit epoch-tracker limitation and ensure operators report promptly after any flow that can leave a non-zero idle baseline.

Developer Response

Fixed in ae8a71f

2.8.5 `RISExLiquidationStrategy.sellToBook()` does not enforce its documented reduce-only invariant

The liquidation strategy documents `sellToBook()` as a reduce-only unwind path, but the function forwards arbitrary order parameters without checking that the `OrderFlags.REDUCE_ONLY` bit is set. Because the call is still `onlyManagement`, this is best classified as a low-severity privileged-hardening issue rather than a broadly exploitable bug.

Technical Details

`RISExLiquidationStrategy.sellToBook()` is documented as: "Sell position to orderbook" and "Places a reduce-only order to close position".
However, the implementation does not enforce that invariant:

```
function sellToBook(IRISExAdapter.OrderRequest calldata request)
    external
    onlyManagement
    returns (bytes32 orderRef)
{
    orderRef = i_adapter.placeOrder(request);
    emit PositionSoldToBook(request.marketId, orderRef, request.size, request.limitPrice
);
}
```

`IRISExAdapter.OrderRequest.flags` carries the order flags, and the repo exposes the relevant bit constant as `OrderFlags.REDUCE_ONLY`. However, `sellToBook()` never checks that bit. The current tests cover the happy path where `flags = OrderFlags.REDUCE_ONLY` in `test_sellToBook_executesSuccessfully()`, while the negative PoC shows that `flags = 0` is also accepted.
As written, management can submit a non-reduce-only order through the liquidation strategy and use this unwind-specific entrypoint to open or increase directional exposure instead of merely closing acquired positions.
PoC

```
function test_poc_sellToBook_doesNotEnforceReduceOnly() public {
    uint256 depositAmount = 100_000e6;
    depositToLiq(user, depositAmount);

    bytes32 expectedRef = bytes32(uint256(67890));
```

```
6       mockPlaceOrder(expectedRef);
7
8       IRISExAdapter.OrderRequest memory request = IRISExAdapter.OrderRequest({
9           marketId: BTC_MARKET_ID,
10          side: IRISExAdapter.Side.Sell,
11          size: 500e6,
12          limitPrice: 50_000e6,
13          flags: 0,
14          extraData: ""
15      });
16
17      vm.prank(management);
18      bytes32 orderRef = liqStrategy.sellToBook(request);
19
20      assertEq(orderRef, expectedRef, "non-reduce-only orders are accepted");
21  }
```

Impact

Low. This is a privileged path guarded by `onlyManagement`, so it does not create a permissionless exploit. The main risk is operational or trust-model drift: an incorrect management action, automation bug, or compromised management key can use the liquidation strategy in a way that violates its intended unwind-only behavior.

Recommendation

Enforce the `OrderFlags.REDUCE_ONLY` bit inside `sellToBook()`.
If the function is truly intended to be unwind-only, the contract should reject any non-reduce-only order at the boundary rather than relying on off-chain discipline or caller conventions.

Developer Response

`sellToBook()` removed from strategies in commit 306716d

2.8.6 `RISExStrategy.placeOrder()` remains able to open or increase positions after emergency shutdown

Emergency shutdown currently prevents new deposits but does not prevent management from placing fresh non-reduce-only orders. The strategy can therefore continue increasing directional exposure even after the emergency role has attempted to freeze operations.

Technical Details

`TokenizedStrategy.shutdownStrategy()` is a one-way emergency switch. After shutdown, deposits and mints are blocked and the strategy can be unwound via emergency maintenance paths.
However, `RISExMarketMakingStrategy.placeOrder()` does not check `TokenizedStrategy.isShutdown()` before forwarding a new order to the adapter.
The only gating logic in `_validateOrderAllowed()` is based on current equity and margin ratio:

- if `equity <= 0`, only `reduceOnly` orders are blocked from opening new risk,
- if `equity > 0`, any non-`reduceOnly` order is allowed whenever the current margin ratio is above `positionLimitBps`.

As a result, once `shutdownStrategy()` has been triggered by the emergency role, `management` can still submit fresh non-`reduceOnly` orders and increase exposure. This weakens the practical meaning of the emergency switch for a perpetuals strategy: the contract stops taking new deposits, but it does not stop taking new directional risk.

PoC

```
function test_poc_shutdown_doesNotBlockNewOrders() public {
    deposit(user, 10_000e6);
    mockAccountEquity(10_000e18);
    mockWithdrawableAmount(10_000e18);

    bytes32 expectedRef = bytes32(uint256(12345));
    vm.mockCall(adapterAddr, abi.encodeWithSelector(IRISExAdapter.placeOrder.selector),
abi.encode(expectedRef));

    vm.prank(emergencyAdmin);
    iStrategy.shutdownStrategy();
    assertTrue(iStrategy.isShutdown(), "strategy should be shut down");

    IRISExAdapter.OrderRequest memory request = IRISExAdapter.OrderRequest({
        marketId: 1,
        side: IRISExAdapter.Side.Buy,
        size: 1e18,
        limitPrice: 50_000e18,
        flags: 0,
        extraData: ""
    });

    vm.prank(management);
    bytes32 orderRef = strategy.placeOrder(request);

    assertEq(orderRef, expectedRef, "new orders still go through after shutdown");
}
```

Impact

Low. An emergency shutdown does not fully freeze risk creation. A compromised or simply uncoordinated management key can reopen or increase positions after shutdown, undermining the emergency admin's attempt to stabilize the strategy and prepare it for withdrawal or unwind.

Recommendation

Gate `placeOrder()` on shutdown state.

The safest default is to reject all new orders when `TokenizedStrategy.isShutdown()` is true. If post-shutdown maintenance is still required, allow only explicitly unwind-oriented operations such as `reduceOnly` orders and continue blocking any order that can increase net exposure.

Developer Response

`placeOrder()` removed from strategies in commit 306716d

2.9 Gas Savings Findings

None.

2.10 Informational Findings

2.10.1 Vault should enforce default withdrawal queue order

The Yearn V3 vault allows users to specify a custom withdrawal queue (`strategies[]` parameter on `withdraw` / `redeem`). If `use_default_queue` is not set to `true` , users can choose which strategy to withdraw from first. This lets users bypass the intended withdrawal order, potentially draining the RLP strategy's small TVL and weakening the protocol's liquidation capacity.

Technical Details

The Yearn V3 vault's `_redeem` function:

```
_strategies: DynArray[address, MAX_QUEUE] = self.default_queue

# If a custom queue was passed, and we don't force the default queue.
if len(strategies) != 0 and not self.use_default_queue:
    _strategies = strategies
```

When `use_default_queue = False` (the default), any user can call `vault.withdraw(amount, receiver, owner, maxLoss, [rlpStrategy])` to withdraw exclusively from the RLP strategy, bypassing MM entirely.
The intended allocation is 95% MM / 5% RLP. The MM strategy has higher liquidity, lower risk of losses (market-making vs holding toxic liquidation positions), and a larger TVL to absorb withdrawals. The RLP strategy holds a small TVL that directly determines its daily liquidation capacity via TVL-based caps (20% per-market, 50% global).
If users withdraw from RLP first:

- RLP TVL shrinks → daily caps shrink proportionally → fewer/smaller liquidations possible
- The protocol's ability to absorb bankrupt positions is weakened at precisely the time when liquidations are most needed (market downturns that also trigger user withdrawals)
- The 95/5 ratio skews, requiring manual `update_debt` rebalancing by the role manager

Impact

Informational. The vault contract is out of scope, but the withdrawal queue configuration directly affects the in-scope strategies' liquidity, epoch tracking, and liquidation capacity.

Recommendation

Set `use_default_queue = true` on the vault and configure the default queue as `[RISExMarketMakingStrategy, RISExLiquidationStrategy]` . This ensures:

1. Withdrawals pull from MM first (larger, more liquid, lower risk)
2. RLP is only tapped when MM cannot cover the full withdrawal
3. RLP TVL is preserved for liquidation capacity
4. Users cannot selectively drain either strategy

This should be part of the vault configuration script (`03_ConfigureVault.s.sol`).

Developer Response

Fixed in commit f1bf988

2.11 Final Remarks

The code is sound when adapter reads are healthy and strategy accounting is refreshed regularly through reports. The key architectural tension is between view-function resilience through `try/catch` fallbacks and correctness in state-changing paths: execution must fail closed on bad data. The intended operating model should be explicit. When the adapter is healthy and the strategy is live, the full logic based on live equity, withdrawable liquidity, and configured caps should be used. When the adapter is unhealthy and the strategy is not shut down, the system should fail closed rather than rely on guessed valuation or guessed withdrawability. A significant portion of the effective risk surface sits in the out-of-scope RISEx adapter; the strategies are well-structured but fundamentally dependent on correct adapter behavior for fund safety, equity accounting, and order management.