



August 2026

Prepared for
VFAT

Audited by
fedebianu
adriro

Sickle Automation

Smart Contract Security Assessment

Contents

1	Review Summary	3
1.1	Protocol Overview	3
1.2	Audit Scope	3
1.3	Risk Assessment Framework	3
1.3.1	Severity Classification	4
1.4	Key Findings	4
1.5	Overall Assessment	5
2	Audit Overview	5
2.1	Project Information	5
2.2	Audit Timeline	5
2.3	Audit Resources	5
2.4	Critical Findings	7
2.5	High Findings	7
2.6	Medium Findings	7
2.6.1	User-controlled callbacks can attack other operations in shared automation batches	7
2.6.2	Pangolin rewards are irrecoverably routed to strategy contracts	9
2.6.3	Shift rebalance targets can remain immediately eligible, forcing repeatable same-range rebalances	12
2.7	Low Findings	17
2.7.1	Manual cross-pool rebalance migrates stale automation policy	17
2.7.2	Shadow multi-compound can penalize pre-existing xShadow	18
2.7.3	V3 multi-compound source configurations do not bind the destination pool	19
2.7.4	No-op deposits leave the ERC-20 compound target unbound from the zap pool	22
2.7.5	NFT claim farm is not bound to the owner's automation opt-in	23
2.7.6	Partial rebalance can orphan the original NFT after settings migration	25
2.7.7	Automated compound can override the configured unstaked custody mode	26
2.7.8	Generic <code>executeFor()</code> rebalances skip vault safeguards	28
2.7.9	Gas-account mode can be disabled after execution but before reimbursement	29
2.7.10	Rollup fee oracles are queried with inner strategy calldata rather than the priced transaction	30
2.7.11	Batched gas-account reimbursements duplicate transaction-wide fixed fees	31
2.7.12	Compound targets remain usable after an NFT leaves Sickle custody	32
2.7.13	Automated rebalances can remove liquidity from an unauthorized NFT	33
2.7.14	Automated exits and rebalances do not bind the removal source to the authorized NFT	34
2.7.15	The Arbitrum gas adapter cannot accurately price the variable L1 poster fee	35
2.7.16	A stale multi-compound target can receive rewards after ownership changes	36
2.8	Gas Savings Findings	38
2.9	Informational Findings	38
2.9.1	Action registrations do not enforce their single-permission-bit invariant	38
2.9.2	Owner-configured rebalance loss limits are not enforced	40
2.9.3	Equalizer NFPG registry accepts non-canonical targets with copied metadata	41
2.9.4	Rebalance can strand NFT automation revocations on obsolete token IDs	42
2.9.5	Automation unnecessarily exposes its internal batching helper	43
2.9.6	Wildcard permissions silently expand when future actions are introduced	44

2.9.7	Automation leaves economic execution choices to trusted keepers	45
2.9.8	Some automated NFT harvest paths use the manual fee selector	46
2.9.9	Automated NFT deposits can add vault funds to an unrelated position manager	47

1 Review Summary

1.1 Protocol Overview

This protocol lets approved automators act on Sickle positions on behalf of their owners. **Automation** exposes two dispatch routes: fixed typed entrypoints for harvest, compound, exit, rebalance and dust deposit, and a generic **executeFor** where an admin binds a (strategy, selector) pair to a permission bit. Every action requires both the automator's permission and the Sickle owner's consent, plus a per-position opt-in stored in **NftSettingsRegistry** or in a pluggable **ISettingsModule** such as **MultiCompoundSettingsRegistry**. **NftFarmStrategy** drives the NFT position lifecycle with target geometry derived by **ShiftMath**, **MultiFarmStrategy** consolidates rewards from many positions into one destination, and **GasAccount** with the per-chain **GasCostAdapter** reimburses keepers from a prepaid native-token balance.

1.2 Audit Scope

This audit covers 16 smart contracts totaling approximately 2050 lines of code across 8 days of review.

```
./contracts
├── interfaces
│   ├── libraries
│   │   └── IShiftMath.sol
│   ├── IGasCostAdapter.sol
│   ├── IMultiAutomation.sol
│   ├── INftAutomation.sol
│   └── ISettingsModule.sol
├── libraries
│   ├── AutomationPermissions.sol
│   └── ShiftMath.sol
├── strategies
│   ├── MultiFarmStrategy.sol
│   └── NftFarmStrategy.sol
├── structs
│   ├── NftSettingsStructs.sol
│   └── ShiftConfigStructs.sol
├── Automation.sol
├── GasAccount.sol
├── GasCostAdapter.sol
├── MultiCompoundSettingsRegistry.sol
└── NftSettingsRegistry.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	3
■ Low	16
■ Informational	9

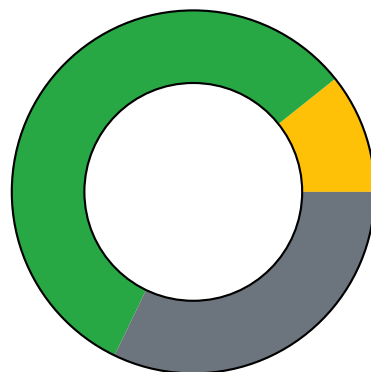


Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

No critical or high severity issues were identified. The recurring theme instead is that the position an entrypoint validates is often not the asset the operation acts on: opt-ins bind an action's source more tightly than its destination or its removal source, safeguards are attached to a specific entrypoint rather than to the operation itself, and user-registered callbacks run inside shared keeper batches. A smaller cluster concerns gas accounting, which reimburses per operation costs that the chain charges per transaction.

2 Audit Overview

2.1 Project Information

Protocol Name: VFAT

Repository: <https://github.com/vfat-io/sickle-contracts>

Commit Hash: [221a7d22fcc7a5fe2fb5ecc9e9ffe493b2353d7c](#)

2.2 Audit Timeline

The audit was conducted from July 27 to August 5, 2026.

2.3 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Average	Authorization is layered and fail-closed, requiring the automator's permission, the owner's consent and a per-position opt-in. The weakness is enforcement rather than model, and it is systematic: most findings validate one position while acting on parameters never bound to it.
Mathematics	Average	Little arithmetic overall, but the rebalance tick geometry holds the only mathematical issue. The remainder is gas accounting, allocating per-transaction costs per batch item.
Complexity	Average	Each piece reads well alone, but the system carries two dispatch routes to the same strategy functions, optional callbacks, pluggable settings modules and per-chain fee adapters. EIP-170 pressure also splits the rebalance checks across a call boundary.
Libraries	Good	Solid base: Solmate, OpenZeppelin clones and an internal library pattern that keeps strategies under the bytecode limit.
Decentralization	Average	Configuration sits behind plain admin roles: the automator allowlist, the action registry, and the caller and target allowlists, with the timelock covering only settings modules. Nearly every finding requires an authorized automator, so the practical security boundary is the keeper.
Documentation	Average	In-code documentation is unusually thorough and often explains rationale rather than restating the code. The weakness is accuracy drift: some comments assert security properties the implementation does not provide.
Testing and verification	Good	The suite is extensive and fork-based, with per-integration harnesses across many chains. The gaps fall in one place: cases are overwhelmingly single-entity, whereas most findings need a second position, a second NFT, or the other dispatch route to surface.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 User-controlled callbacks can attack other operations in shared automation batches

`Automation.setNftRebalanceCallback()` and `Automation.setNftDepositCallback()` allow each Sickle owner to register arbitrary callback code. Because Automation executes these callbacks inside keeper batches without isolating different owners, a malicious callback can manipulate shared markets and vault state around unrelated users' operations, extract funds from vault shareholders, or make the complete batch revert after consuming keeper gas.

Technical Details

The callback setters only require `msg.sender` to equal `sickle.owner()`. They do not restrict callbacks to approved implementations or bind a callback to the integration operated by its Sickle. The authorization performed by Automation confirms that the outer caller is an approved automator and that each Sickle consented to the action, but it does not constrain the external actions performed by a callback.

`Automation.rebalanceFor()` uses three global phases. It first invokes every `beforeRebalance()` callback, then executes every strategy operation through `multicall()`, and finally invokes every `onRebalance()` callback. A funded malicious callback can therefore trade against a pool or route in `beforeRebalance()`, leave the manipulated state in place while unrelated users rebalance, and reverse the trade in `onRebalance()`. This embeds an atomic sandwich inside the trusted keeper transaction, including when the transaction is submitted privately. The victims lose value through adverse swap execution or liquidity placement, while the attacker retains the value captured by the reverse trade. Keeper-supplied minimum outputs and liquidity minimums bound the attack but do not prevent extraction within the accepted tolerance.

The global phases also allow malicious callbacks to interleave with `NftFarmVault`'s security lifecycle. `NftFarmVault.beforeRebalance()` snapshots an absolute pre-rebalance value V , and `NftFarmVault.onRebalance()` later compares the replacement value against that snapshot. The `nonReentrant` guard covers each callback separately and is released between them. If a later attacker pre-hook calls the vault's public `deposit()` with value D after the snapshot, a rebalance that incorporates the deposit and loses L is accepted whenever $V + D - L \geq V * (1 - \text{maxLoss})$. The actual permitted loss is therefore $D + \text{maxLoss} * V$ instead of $\text{maxLoss} * V$. The callback can combine this stale snapshot with the rebalance sandwich, capture L , and redeem its fairly minted shares after the victim's post-hook. For example, with $V = 100$, a 1% loss limit, $D = 4$, and $L = 5$, the final value of 99 passes the nominal 1% floor. After redeeming, the attacker can retain approximately 4.81 units taken from the original shareholders.

Exact-liquidity rebalances permit a stronger share inflation variant. Consider a batch ordered as attacker item A, victim vault, and attacker item B, where both attacker items use the same callback contract. The victim pre-hook first snapshots V , then B's pre-hook makes a small proportional deposit D into the old NFT. If the keeper encoded the old NFT's original exact liquidity, the connector removes only that amount. `UniswapV3Connector.removeLiquidity()` does not burn the old NFT because the callback-added liquidity remains, while the strategy creates a replacement NFT containing approximately V .

A's post-hook executes before the victim post-hook and can call `deposit()` again with value E . Vault storage still references the old NFT, so share issuance is priced against the small residual D and any idle pool-token balance while the replacement NFT is omitted. The callback can consequently mint approximately E / D times the existing share supply. The victim post-hook then resolves the replacement NFT, verifies its value against the original snapshot, and updates the stored token ID. B's post-hook can immediately redeem the inflated shares from the replacement. With $V = 100$, $D = 0.01$, and $E = 0.99$, the attacker can obtain approximately 99% of the shares and redeem about 99 units after contributing only 1 unit. This variant requires exact-liquidity removal, an in-range and TWAP-valid residual position, suitable resolver and custody behavior, low idle balances, and deposit-cap headroom. Using `type(uint128).max` removal consumes the callback-added seed and prevents this particular transition, but leaves the stale loss-floor attack described above.

`Automation.depositFor()` executes each item as `beforeDeposit()` followed by its strategy deposit. Two attacker-controlled entries can bracket victim entries because the function does not reject duplicate Sickles or isolate owners. The first callback can manipulate a shared pool, the victim deposits can then swap or add liquidity at the manipulated state, and the second callback can reverse the trade. The attacker Sickles can use negligible operations in unrelated pools because their callbacks are not restricted to the corresponding positions. Victim loss is again bounded only by the keeper's execution minimums.

Finally, either callback can deliberately revert, exhaust the remaining gas, or perform expensive external work before returning. Any revert rolls back the complete mixed-user batch, and both failed and successful expensive callbacks charge gas to the keeper. One malicious job can therefore block unrelated jobs or repeatedly impose transaction costs until the keeper identifies and excludes it. There is no callback gas limit or failure isolation.

Impact

Medium. A malicious Sickle owner included in a mixed keeper batch can extract value from unrelated Sickles and vault shareholders, bypass a vault's configured rebalance loss limit, and under narrower exact-liquidity conditions inflate shares and drain most of a replacement position. The attacker can also revert complete batches or impose arbitrary gas costs on the keeper. The attacks require attacker-controlled jobs to be co-batched with victims, and the strongest paths additionally depend on ordering, liquidity-removal parameters, position state, and vault cap headroom.

Recommendation

Restrict callback registration to approved, immutable implementations, or execute callback-enabled jobs from different owners in separate transactions. For `rebalanceFor()`, cache the callback address and execute each item's `beforeRebalance()`, strategy operation, and `onRebalance()` as one isolated lifecycle. Reject duplicate or adversarially ordered entries where isolation is not guaranteed.

Add a vault-level rebalance lock that remains active from `beforeRebalance()` until `onRebalance()` and rejects deposits and redemptions during that interval. Also require the

share supply to remain unchanged, ensure the old NFT is fully consumed, and price deposits against every vault asset, including any in-flight replacement NFT. If arbitrary callbacks must remain supported, apply an explicit gas limit and isolate callback failure per job rather than allowing one callback to affect an unrelated batch. Do not silently ignore a failed vault callback because the callback enforces required safety checks.

Developer Response

Acknowledged, no code change for this reason. We control batch composition — an attacker cannot place themselves next to a victim, so this is not attacker-reachable in practice.

- Note that restricting who may register a callback does not address this: the callback contract can own the Sickle and register itself, so `msg.sender == callback == owner` passes every check.
- Separately, in [vfat-io/sickle-contracts#906](#) (9183d7b6), `rebalanceFor()` now brackets each Sickle's callbacks around its own rebalance instead of running three global phases. We did that for consistency with `depositFor()`, which already worked that way, not as a mitigation.

2.6.2 Pangolin rewards are irrecoverably routed to strategy contracts

`PangolinV3Connector.claim()` passes `msg.sender` as the PNG recipient. Because the connector executes by `delegatecall` from a Sickle, `msg.sender` is the calling strategy rather than the Sickle, so claimed rewards accumulate at the strategy contract and cannot be processed or swept for the user.

This is a technically valid Medium-severity bug, but its root cause and required fix are in `PangolinV3Connector`, which is outside the listed audit scope.

Technical Details

`NftFarmStrategy._claim()` and `MultiFarmStrategy._harvestNftPositions()` resolve the relevant farm connector and ask the Sickle to execute its `claim()` function through `Sickle.multicall()`. The Sickle uses `delegatecall`, so connector code runs with the Sickle as `address(this)` while preserving the strategy contract as `msg.sender`.

`PangolinV3Connector.claim()` first reads the position's accrued `rewardOwed`. When the value is non-zero, it calls:

```
1 IPangolinPositionManager(address(position.nft)).claimReward(  
2     position.tokenId, msg.sender  
3 );
```

The external call originates from the Sickle and therefore passes the position manager's ownership authorization, but the independently supplied recipient is the strategy address. The position's PNG entitlement is consumed and the tokens are transferred to `NftFarmStrategy` or `MultiFarmStrategy`, not to the Sickle.

Subsequent fee, zap, and sweep `delegatecalls` operate in the Sickle context and inspect only the Sickle's balances. They cannot process PNG held directly by the strategy. Neither strategy exposes a token-recovery function, so the affected claim can complete and emit normal events while its PNG remains inaccessible to the user.

Proof of concept

The following production-fork replay uses a real Pangolin position immediately before an observed reward claim. It does not seed balances, storage, or reward entitlement.

```

1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.17;

4  import { Test } from "forge-std/Test.sol";
5  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

7  import { ConnectorRegistry } from "contracts/ConnectorRegistry.sol";
8  import {
9      IPangolinPositionManager
10 } from "contracts/connectors/pangolin/PangolinV3Connector.sol";
11 import {
12     INonfungiblePositionManager
13 } from "contracts/interfaces/external/uniswap/INonfungiblePositionManager.sol";
14 import { NftFarmStrategy } from "contracts/strategies/NftFarmStrategy.sol";
15 import { Farm } from "contracts/structs/FarmStrategyStructs.sol";
16 import {
17     NftPosition,
18     SimpleNftHarvest
19 } from "contracts/structs/NftFarmStrategyStructs.sol";

21 contract PangolinRewardRecipientPoC is Test {
22     uint256 private constant FORK_BLOCK = 85_601_329;

24     address private constant OWNER =
25         0x8E71cE3a15B5b1092983123b40e88fC1F8E442bA;
26     address private constant SICKLE =
27         0x4f123651921834a6b4582714e6395D42737C5c8d;
28     address private constant STRATEGY =
29         0x2E075EdF77ac8dF12B88F22170bB064c2FFbED44;
30     address private constant MANAGER =
31         0xf40937279F38D0c1f97aFA5919F1cB3cB7f06A7F;
32     address private constant CONNECTOR =
33         0x06E2eaEa943cACc9c39D170330219218C4F34593;
34     address private constant CONNECTOR_REGISTRY =
35         0x114F84Dc0a203832D23C4f497ce91Fa7dc8B8F16;

37     address private constant PNG =
38         0x60781C2586D68229fde47564546784ab3fACA982;
39     address private constant WAVAX =
40         0xB31f66AA3C1e785363F0875A1B74E27b85FD66c7;
41     address private constant USDC =
42         0xB97EF9Ef8734C71904D8002F8b6Bc66Dd9c48a6E;
43     uint256 private constant TOKEN_ID = 13_668;

45     function test_PangolinRewardIsSentToStrategy() public {
46         vm.createSelectFork(vm.envString("AVAX_RPC_URL"), FORK_BLOCK);

48         assertEq(
49             ConnectorRegistry(CONNECTOR_REGISTRY).connectorOf(MANAGER),
50             CONNECTOR
51         );
52         assertEq(
53             INonfungiblePositionManager(MANAGER).ownerOf(TOKEN_ID), SICKLE
54         );

56         (,,, uint256 rewardBefore) =
57             IPangolinPositionManager(MANAGER).positionReward(TOKEN_ID);
58         assertGt(rewardBefore, 0);

```

```

60     uint256 strategyPngBefore = IERC20(PNG).balanceOf(STRATEGY);
61     uint256 sicklePngBefore = IERC20(PNG).balanceOf(SICKLE);
62     uint256 ownerPngBefore = IERC20(PNG).balanceOf(OWNER);

64     address[] memory rewardTokens = new address[](2);
65     rewardTokens[0] = WAVAX;
66     rewardTokens[1] = USDC;

68     NftPosition memory position = NftPosition({
69         farm: Farm({ stakingContract: MANAGER, poolIndex: 0 }),
70         nft: INonfungiblePositionManager(MANAGER),
71         tokenId: TOKEN_ID
72     });
73     SimpleNftHarvest memory harvest = SimpleNftHarvest({
74         rewardTokens: rewardTokens,
75         amount0Max: type(uint128).max,
76         amount1Max: type(uint128).max,
77         extraData: ""
78     });

80     vm.prank(OWNER);
81     NftFarmStrategy(payable(STRATEGY)).simpleHarvest(position, harvest);

83     (,,, uint256 rewardAfter) =
84         IPangolinPositionManager(MANAGER).positionReward(TOKEN_ID);
85     uint256 strategyGain =
86         IERC20(PNG).balanceOf(STRATEGY) - strategyPngBefore;

88     assertGt(strategyGain, 0);
89     assertEq(rewardAfter, 0);
90     assertEq(IERC20(PNG).balanceOf(SICKLE), sicklePngBefore);
91     assertEq(IERC20(PNG).balanceOf(OWNER), ownerPngBefore);
92 }
93 }

```

Run with:

```

1 AVAX_RPC_URL=https://api.avax.network/ext/bc/C/rpc forge test --match-path test/poc/
  PangolinRewardRecipient.t.sol --match-test test_PangolinRewardIsSentToStrategy -vv

```

The claim clears the real pending entitlement, increases the deployed strategy's PNG balance, and leaves both the Sickle and owner PNG balances unchanged.

Impact

Medium, out of scope. Each affected Pangolin claim can consume the position's entire accrued PNG entitlement while delivering none of it to the Sickle or its owner. The behavior is deterministic during ordinary Pangolin harvests and can recur for future rewards.

The position NFT, deposited liquidity, collected pool tokens, and other reward assets remain safe. The loss is limited to accrued Pangolin yield rather than position principal, so High severity is overstated. The finding is excluded from the in-scope candidate set because `contracts/connectors/pangolin/PangolinV3Connector.sol` is absent from `scope.txt`.

Recommendation

Pass `address(this)` rather than `msg.sender` as the reward recipient. Under `delegatecall`, `address(this)` is the Sickle:

```

1 - positionManager.claimReward(position.tokenId, msg.sender);
2 + positionManager.claimReward(position.tokenId, address(this));

```

Disable or remap the affected connector until the fix is deployed. Add an integration test that invokes the connector through `Sickle.multicall()` and asserts that the Sickle's PNG balance increases while the strategy and connector balances remain zero.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 5b302195.

- `PangolinV3Connector.claim()` passed `msg.sender`, which under `delegatecall` is the calling strategy. Now `address(this)`, matching the `_collect()` call two lines above.
- 76.6 PNG on `NftFarmStrategy` and 861.3 PNG on `MultiFarmStrategy` were already stranded on Avalanche when we fixed this, and are unrecoverable.
- Our suite missed it because `PangolinV3Pools.t.sol` declares `address(0)` as reward token, which skips 16 harness tests including the one asserting the owner's balance grows. Added a reward-earning sibling suite in #897 (a9d6bc44).
- PR #900 (8f1efa4c) reverts an `earned()` override we added alongside that fix — it over-reported pending rewards by 108x against a real block, so we took it out rather than ship a wrong number to the lens and keeper.

2.6.3 Shift rebalance targets can remain immediately eligible, forcing repeatable same-range rebalances

`ShiftMath._formulaTicksForRebalance()` derives the exact range an automator must mint, but never checks that the resulting position clears the user's rebalance trigger. With a supported negative buffer the target is a fixed point: the same position is withdrawn and re-minted to identical ticks at an unchanged pool tick. Because the shift check rejects every range except the formula's output, the automator cannot avoid this by choosing a better target.

Technical Details

`NftSettingsRegistry.validateRebalanceFor()` permits a rebalance whenever the tick lies outside the buffered interval

`[tickLower - bufferTicksBelow, tickUpper + bufferTicksAbove):`

```

1 // contracts/NftSettingsRegistry.sol:182-188
2 if (
3     poolInfo.tick
4     >= positionInfo.tickLower - int24(config.bufferTicksBelow)
5     && poolInfo.tick
6     < positionInfo.tickUpper + int24(config.bufferTicksAbove)
7 ) {
8     revert TickWithinRange();
9 }

```

Negative buffers are a documented feature — `RebalanceConfig` states *"Can be negative (rebalance before position goes above range)"* — and `_checkRebalanceConfig()` bounds them only by $\pm 2 * \text{MAX_TICK}$. They pull the trigger *inside* the position, so a position can be eligible while the tick still sits within its own range.

`preMintRebalanceChecks()` re-derives that trigger to choose a direction, then computes the target with formulas that only preserve the old width and offset the range by the signed entry value:

```
1 // contracts/libraries/ShiftMath.sol:182-204
2 expectedLower = tickFloor - cfg.entryTickSpacesUp * spacing;
3 expectedUpper = expectedLower + width;

5 // or

7 expectedUpper = tickFloor + (cfg.entryTickSpacesDown + 1) * spacing;
8 expectedLower = expectedUpper - width;
```

Neither `_formulaTicksUp()` nor `_formulaTicksDown()` checks whether the tick falls inside the *new* buffered interval. On the WBTC/USDC 0.3% pool used by the existing fork suite — `tickSpacing = 60`, width 3 tick-spaces, `entryTickSpacesDown = 1`, `bufferTicksBelow = -120`:

	old position	after shift
range	[70800, 70980)	[70680, 70860)
stay-put interval	[70920, 70980)	[70800, 70860)
tick 70785	outside → eligible	outside → still eligible

The replacement's own lower trigger is `70680 + 120 = 70800`, and `70785 < 70800`, so it is eligible the moment it is minted — while the tick sits inside its range, earning fees. Re-running the formula against `[70680, 70860)` returns `[70680, 70860)`.

The numbers are not special. With `W` the width in tick-spaces and `r = tick - floor(tick, ts)`, a shifted-down target stays past its own lower trigger whenever `bufferTicksBelow < (down + 1 - W) * ts - r`, with the symmetric relation for the up direction; `-120` satisfies it for every `r`. With zero buffers the relation reduces to an entry value that places the position entirely outside the tick, and `ShiftConfig` accepts every signed `int24` for both entry fields and is validated nowhere.

Such configurations matter because shift removes the automator's discretion over the target. Without a shift payload, `_shiftSetupInner()` returns `isShift = false` and the keeper picks the new ticks itself, so a reasonable one chooses a range that clears the trigger. With shift enabled, `ShiftMismatch` rejects any range other than the formula's output. The automator's only options become "mint the retriggering range" or "do not rebalance at all".

Nothing interrupts the resulting cycle. `NftFarmStrategy._rebalance()` mints a new NFT and migrates the settings through `transferNftSettings()`, preserving the exact condition that caused the first rebalance; `_checkTickWidth()` re-runs on migration but passes, because the formula preserves the width. `postMintRebalanceChecks()` verifies only the minted pool identity and exact tick equality. `RebalanceConfig.delayMin` ("Delay in minutes before rebalancing") exists as a struct field and `NatSpec` but no contract reads it, so there is no on-chain cooldown either.

Proof of concept

Add both of the following to `test/strategies/ShiftRebalance.t.sol`, the existing WBTC/USDC mainnet-fork suite. The test drives two consecutive `Automation.rebalanceFor()` calls through the approved automator and asserts the pool tick never moves between them. The helper wraps the file's existing `_buildShiftRebalance()` to

override the mock-swap input and the mint caps, so the second call can be funded from what the first left; the geometry it builds is unchanged.

```

1  function _buildShiftRebalanceSized(
2      uint256 tokenId,
3      int24 newLower,
4      int24 newUpper,
5      uint256 swapAmountIn,
6      uint256 amount0Desired,
7      uint256 amount1Desired
8  ) private returns (NftRebalance memory params) {
9      params = _buildShiftRebalance(tokenId, newLower, newUpper);
10     params.increase.zap.swaps[0].amountIn = swapAmountIn;
11     params.increase.zap.addLiquidityParams.amount0Desired = amount0Desired;
12     params.increase.zap.addLiquidityParams.amount1Desired = amount1Desired;
13 }

15 function test_shift_same_range_rebalance_repeats() public prank {
16     bytes memory extraData = abi.encode(
17         ShiftConfig({
18             magic: SHIFT_MAGIC,
19             enabled: true,
20             entryTickSpacesUp: 1,
21             entryTickSpacesDown: 1
22         })
23     );
24     int24 spacing = contracts.pool.tickSpacing();
25     uint256 tokenId = _depositAndOptIn(extraData, -2 * spacing, 0);

27     (int24 oldLower, int24 oldUpper,) = _get_range_and_liquidity(tokenId);
28     int24 width = oldUpper - oldLower;

30     _pushTickBelow(oldLower);
31     int24 tick = _get_tick(contracts.pool);

33     int24 expectedUpper = _floorToSpacing(tick, spacing) + 2 * spacing;
34     int24 expectedLower = expectedUpper - width;

36     vm.stopPrank();
37     vm.startPrank(ctx.approvedAutomator);

39     _callRebalanceFor(
40         _buildShiftRebalanceSized(
41             tokenId, expectedLower, expectedUpper, 0.05e8, 0.1e8, 10_000e6
42         )
43     );
44     uint256 secondTokenId = _get_token_id();
45     assertGt(secondTokenId, tokenId, "first rebalance did not mint");

47     (int24 l2, int24 u2,) = _get_range_and_liquidity(secondTokenId);
48     assertEq(l2, expectedLower);
49     assertEq(u2, expectedUpper);

51     // The zap swaps through the mock router, so the real pool tick is
52     // untouched: the second call sees exactly the state the first left.
53     assertEq(_get_tick(contracts.pool), tick, "pool tick moved");

55     _callRebalanceFor(
56         _buildShiftRebalanceSized(
57             secondTokenId, expectedLower, expectedUpper,
58             0.01e8, 0.02e8, 2000e6
59         )
60     );

```

```

61     uint256 thirdTokenId = _get_token_id();
62     assertGt(thirdTokenId, secondTokenId, "second rebalance did not mint");

64     (int24 l3, int24 u3,) = _get_range_and_liquidity(thirdTokenId);
65     assertEq(l3, l2, "lower tick changed");
66     assertEq(u3, u2, "upper tick changed");
67 }

```

Impact

Medium. An honest automator scanning for eligible positions can re-execute the same rebalance because each successful call leaves the next one valid, and the shift constraint denies it any alternative target. Every subsequent iteration withdraws and re-mints the full position while ending at the same ticks from which that iteration started, can charge the configured rebalance fee against the withdrawn principal and `HarvestFor` against accrued rewards, and may incur swap, slippage and prepaid-gas costs. At an unchanged tick the cycle continues until the user disables automation, a cutoff is crossed, the available balance is depleted, or the keeper suppresses another otherwise-valid execution.

Recommendation

Reject the offending configuration rather than the individual rebalance. If a configuration violates the static bounds below, at least one tick remainder produces an immediately eligible target; where the relation holds for every remainder, so does every target calculated at its unchanged source tick. A per-call check therefore makes the affected shift direction unavailable for every tick remainder when the corresponding every-remainder bound is violated, and unavailable only at the offending remainders otherwise — trading a fee drain for an automation outage whenever that direction is triggered.

Validate the shift payload against the buffers and width when settings are stored, in `checkConfigValues` alongside the existing `_checkTickWidth()` — decoding `ShiftConfig` only when `extraData` carries `SHIFT_MAGIC`, and inside the `settings.autoRebalance` branch, where the width in tick-spaces is already available as `tickSpacesBelow + tickSpacesAbove + 1`.

Requiring the target to sit inside its buffered interval for every tick remainder yields four bounds: `bufferTicksBelow >= -up * ts`, `bufferTicksAbove >= (up - W + 1) * ts`, `bufferTicksBelow >= (down + 1 - W) * ts`, and `bufferTicksAbove >= -down * ts`.

Evaluate them in `int256`: unrestricted signed entry values are multiplied by the tick spacing before the invalid configuration is rejected.

As defence in depth for settings stored before that validation exists, add the exact per-call check to `_shiftSetupInner()` immediately after the target is derived, so an unsafe target is rejected before harvest, withdrawal, swaps or minting consume gas:

```

1  int256 lowerTrigger = int256(expectedLower)
2    - int256(settings.rebalanceConfig.bufferTicksBelow);
3  int256 upperTrigger = int256(expectedUpper)
4    + int256(settings.rebalanceConfig.bufferTicksAbove);
5  int256 tick = int256(poolInfo.tick);

7  if (tick < lowerTrigger || tick >= upperTrigger) {
8      revert ShiftTargetStillEligible(
9          poolInfo.tick, expectedLower, expectedUpper
10     );
11 }

```

The widening matters here even though the equivalent addition in `validateRebalanceFor()` cannot overflow: there the operand is a real position tick bounded by `MAX_TICK`, whereas `expectedLower/expectedUpper` are derived from unvalidated signed entry values and are not. This pre-mint guard addresses the fixed point at the sampled tick but cannot guarantee the final-state invariant, since zap swaps may subsequently move the pool tick.

`_verifyShiftTicksInner()` already compares the minted ticks against pre-mint-computed values, so that staleness should not be assumed safe: either derive the target after price-changing swaps, prevent the zap from moving the configured pool, or add a post-mint check as a backstop. A post-mint revert rolls back fees and swaps atomically but burns the gas of the full call, so it should not be the primary validation.

If deliberately out-of-range partial shifts must remain supported, do not relax these checks — instead record the handled trigger on the migrated settings and require demonstrable price progress before the next rebalance. Wiring up the unused `RebalanceConfig.delayMin` would bound the loop rate but would not remove the fixed point.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 496d83c3, rejecting the formula's fixed point (`formula(T) = T`) so a target identical to the position it replaces is refused, which breaks the repeat cycle while leaving the first genuine shift intact.

- The recommended config-time bounds include `bufferTicksBelow >= -up * ts`. With `entryTickSpacesUp = 0` that reduces to `bufferTicksBelow >= 0`.
- `test_shift_lower_last_band_succeeds` (`test/strategies/ShiftRebalance.t.sol:449`) sets `entryTickSpacesUp/Down = 0` and `bufferTicksBelow = oldLower - tick - 1`, which is negative for an in-range position.
- So that configuration violates the bound and would be rejected at write time, which is why we did not take the recommendation — the negative buffer is what makes the trigger fire while the tick is still in range, and that is the behaviour the config is exercising deliberately.

yAudit Follow-up

We agree that the proposed config-time bounds would prohibit intended in-range triggering, so we withdraw that recommendation. However, we do not think 496d83c3 closes every repeat cycle. `_formulaTicksForRebalance()` also selects a direction from `oldPos`, which changes after each rebalance. At a fixed tick, the two possible targets are:

```
1 U = [F - up*ts, F - up*ts + W*ts)
2 D = [F + (down+1-W)*ts, F + (down+1)*ts)
```

These targets are equal only when `W == up + down + 1`. This holds for the original configuration, but it is not enforced generally.

For example:

```
1 tick = 100000, F = 99960, ts = 60
2 W = 2, up = down = 0
3 bufferBelow = -41, bufferAbove = -20

5 D = [99900, 100020), stay = [99941, 100000)
6 tick crosses the upper trigger -> U

8 U = [99960, 100080), stay = [100001, 100060)
9 tick crosses the lower trigger -> D
```

Both stay intervals are non-empty and every target differs from the position it replaces. The sequence therefore remains `D -> U -> D -> ...`, while `ShiftTargetUnchanged` never fires. The current check covers fixed points, but not this two-range cycle. A one-step look-ahead would cover the remaining unchanged-tick case: after deriving the target, project that range as the next `oldPos` and reject the shift if the next applicable target returns to the original range. Non-applicability of the projected position should be treated as a valid terminating shift. At a fixed tick, only `U` and `D` can be produced, so the existing fixed-point check plus this two-cycle check covers every successful repeat cycle without imposing config-time restrictions on intended in-range triggering.

Developer Response

Follow-up fixed in [vfat-io/sickle-contracts#917](https://github.com/vfat-io/sickle-contracts/pull/917), commit `bdf5fa0a` (merge `28df5d08`), building on the fixed-point guard in `496d83c3`.

The guard now structurally rejects direct U-to-D and D-to-U transitions within one tick-space, including raw-tick drift, while preserving negative-buffer in-range shifts. Cross-tick-space cycles requiring negative buffers are explicitly documented as outside this narrow stateless guard.

2.7 Low Findings

2.7.1 Manual cross-pool rebalance migrates stale automation policy

A manual rebalance can mint the replacement NFT in a different pool while automatically preserving the original pool's exit and rebalance settings. Subsequent automation therefore evaluates Pool A's price while operating on the replacement position in Pool B.

Technical Details

`NftFarmStrategy.rebalance()` is an owner-triggered path and intentionally accepts caller-supplied rebalance parameters without the keeper-only target checks used by `rebalanceFor()`. Through the nested add-liquidity parameters, the owner can mint a replacement position in another pool supported by the same NFT manager. Allowing that manual mint is not itself the issue. The inconsistency arises because `_rebalance()` calls `_resetNftSettings()` immediately after the new NFT is minted. `NftSettingsLib.transferNftSettings()` reads the original settings and forwards them unchanged to `NftSettingsRegistry.transferNftSettings()`, which stores them under the newly minted token ID. The migrated values still contain Pool A's `pool`, `poolId`, exit triggers, and rebalance policy even when the new token belongs to Pool B.

The registry does not establish that the new token belongs to the stored pool. When `autoRebalance` is enabled, `_checkTickWidth()` compares the new position's width with Pool A's tick spacing, but it does not derive the new position's pool identity. A Pool B position with a compatible tick spacing and width therefore passes. When only `autoExit` is enabled, even this width check is skipped and no position-to-pool consistency check is performed.

The resulting settings entry combines a Pool B token ID with Pool A policy.

`validateExitFor()` reads Pool A's tick to evaluate the migrated exit thresholds, while `exitFor()` binds liquidity removal to the current token ID in Pool B. Pool B can consequently be exited solely because Pool A crossed the old threshold.

The rebalance path has a stronger consequence. `rebalanceFor()` loads the migrated settings and passes them to the keeper-only checks. `ShiftMath` evaluates Pool A's tick against Pool B's

current range, requires the replacement mint to resolve to the stored Pool A identity, and accepts the final position when that condition holds. A successful automated call can therefore fully remove Pool B's liquidity and remind the proceeds into Pool A.

This requires a cross-pool manual rebalance under the same NFT manager and a farm connector that accepts the replacement position or performs no custody-changing deposit. It can arise from an integration error or from using `rebalance()` for an intentional cross-pool reposition. The separate `move()` path already supports such moves while accepting explicit replacement settings.

Impact

Low. An honest keeper following the migrated on-chain policy can prematurely exit a Pool B position based on Pool A's price or move Pool B's principal back into Pool A. The user retains control of the funds, but incurs unintended swaps, protocol fees, slippage, interrupted yield, and exposure to a pool the manual transaction had replaced. The prerequisite is an owner-triggered cross-pool call through `rebalance()` rather than `move()`, which limits likelihood but does not require a malicious keeper payload.

Recommendation

Before migrating settings, construct the old position's `NftKey` and load its settings through `nftSettingsRegistry.getNftSettings(key)`. After `_zapIn()` mints the replacement, retrieve `newTokenId` before calling `_resetNftSettings()`. If `autoExit` or `autoRebalance` is enabled, call the existing `shiftMath.postMintRebalanceChecks()` with `isShift = false` and zero expected ticks. In this mode, the function executes `_verifyMintPoolIdentityInner()` but skips `_verifyShiftTicksInner()`, providing the required pool-identity check without any new helper or refactoring.

Only call `_resetNftSettings()` after this check succeeds. A pool mismatch will then revert before stale policy is transferred; the owner can instead use `move()` with explicit replacement settings. Limiting the check to positions with `autoExit` or `autoRebalance` preserves unrestricted manual rebalances for positions without pool-sensitive automation. No storage-layout or settings migration change is required.

Developer Response

Fixed in [PR #925](#).

2.7.2 Shadow multi-compound can penalize pre-existing xShadow

`ShadowRewardBehavior.Exit` processes the Sickle's complete xShadow balance rather than only newly claimed rewards. An authorized keeper can therefore apply the exit penalty to xShadow retained from unrelated positions or earlier operations.

Technical Details

The multi-compound strategy allows a Sickle owner to configure automated consolidation of specific farm positions into a pinned destination. Shadow V2 and V3 gauge connectors support three reward behaviors: `Exit` (approximately 50% penalty to liquid SHADOW), `X33`

(no-penalty deposit), and **Keep** (retain as xShadow). The owner binds source and destination farms in `MultiCompoundSettingsRegistry`, and the automated `compoundMultipleFor()` and `nftCompoundMultipleFor()` paths require the owner to grant `MULTI_COMPOUND` permission. When an authorized keeper calls either entry point, `_validateMultiCompoundFor()` validates that every source was opted into automation with the supplied destination farm. The keeper also supplies claim parameters for each source, including opaque `extraData` fields. `ShadowV2GaugeConnector.claim()` and `ShadowV3GaugeConnector.claim()` decode this data as a `ShadowRewardBehavior` enum. `MultiFarmStrategy._harvestErc20Positions()` and `_harvestNftPositions()` forward the keeper-controlled `extraData` directly to the resolved connector through `sickle.multicall()`. The Shadow connectors permit `ShadowRewardBehavior.Exit`. After calling `getReward()`, both `ShadowV2GaugeClaim._claimGaugeRewards()` and `ShadowV3GaugeClaim._claimGaugeRewards()` read the Sickle's complete xShadow balance with `IERC20(X_SHADOW).balanceOf(address(this))`. When the keeper has selected `Exit`, the full balance is passed to `IXShadow.exit(rewards)`, triggering the approximately 50% penalty on the entire amount rather than only the newly claimed rewards for the opted-in source. This behavior applies to xShadow accumulated from other positions, prior manual operations, and any prior **Keep** decisions. The owner's source-specific opt-in and the strategy's source-farm validation do not constrain which xShadow is affected. Because the connectors execute in the Sickle's context via `delegatecall`, `address(this)` resolves to the victim's Sickle, and the balance query captures all xShadow at rest in that account.

Impact

Low. A keeper can consume the Sickle's entire pre-existing xShadow balance through the penalized exit, not merely the rewards of the opted-in source. The approximately 50% penalty applies to the full amount, causing irreversible loss to the user. This affects xShadow the owner previously retained using the **Keep** behavior, including balances from unrelated positions and manual actions. The transaction completes atomically with valid destination deposits and emits successful compound events, giving no indication that a large pre-existing balance was forcibly exited at the penalized rate.

Recommendation

Snapshot xShadow before the first Shadow reward call and apply `Exit` or `X33` only to the positive claim delta. If full-balance conversion remains useful, expose it as a separate owner-only operation rather than claim `extraData` available to automation. Add an automated multi-compound regression test proving that the starting xShadow balance cannot decrease.

Developer Response

Acknowledged but doesn't need fixing as we don't use **Keep** in production when building calldata, and so this hasn't occurred (also being a small dex at this point).

2.7.3 V3 multi-compound source configurations do not bind the destination pool

`MultiFarmStrategy` binds each multi-compound source to a destination through `MultiCompoundConfig.target` and `targetPoolId`. This identifies the target pool for V4-style positions, but not for bare V3 position managers: every pool under the manager shares the

same `Farm`, while the V3 `targetPoolId` is always zero. An authorized keeper can therefore consolidate rewards from a source configured under one V3 pool into an owner-enabled target position belonging to another pool under the same manager.

Technical Details

`nftCompoundMultipleFor()` first validates the keeper-selected deposit position and then passes its `farm` and derived `targetPoolId` to every source check:

```
1 bytes32 targetPoolId = _checkDepositTarget(sickle, params);
2 _validateMultiCompoundFor(
3     sickle,
4     params.claims,
5     params.nftClaims,
6     params.depositPosition.farm,
7     targetPoolId
8 );
```

`_checkDepositTarget()` correctly requires the candidate NFT to have an owner-written opt-in and binds the zap to that candidate's manager and `tokenId`. The issue is not that the keeper can choose an unconfigured or foreign target. It is that `_checkSourceConfig()` relates a source to the candidate only through `(Farm, targetPoolId)`:

```
1 if (
2     config.target.stakingContract != target.stakingContract
3     || config.target.poolIndex != target.poolIndex
4 ) {
5     revert TargetMismatch();
6 }
7 if (config.targetPoolId != targetPoolId) {
8     revert TargetPoolMismatch(config.targetPoolId, targetPoolId);
9 }
```

For V4-style connectors, `_positionPoolId()` obtains a non-zero `PoolId` directly from the position manager. This lets a source configuration distinguish two pools managed by the same contract.

For V3-style connectors, the helper calls `positionPoolKey()` with a zero factory.

`UniswapV3Connector.positionPoolKey()` can derive the position's actual pool address only by querying that factory and always returns a zero `poolId`:

```
1 return NftPoolKey({
2     poolAddress: IUniswapV3Factory(poolFactory).getPool(
3         token0, token1, fee_
4     ),
5     poolId: bytes32(0) // Uniswap V4 only
6 });
```

The zero-factory call reverts and `_positionPoolId()` deliberately catches it and returns `bytes32(0)`. As a result, the pool address recovered by the connector is discarded and all V3 positions are represented by a zero `targetPoolId`.

This collapses the complete source-target key for a bare V3 manager. The Uniswap V3 test fixture uses the same value `Farm{UniswapV3.NFT_MANAGER, 0}` for both the WBTC/WETH and USDC/WETH pools in `test/strategies/UniswapV3.t.sol`. Therefore, these two owner-written configurations are indistinguishable to `_checkSourceConfig()`:

```
1 // A is a WBTC/WETH position; B is a USDC/WETH position.
2 config[A] = MultiCompoundConfig({
3     enabled: true,
```

```

4     target: Farm({ stakingContract: NFT_MANAGER, poolIndex: 0 }),
5     targetPoolId: bytes32(0)
6 });
7 config[B] = MultiCompoundConfig({
8     enabled: true,
9     target: Farm({ stakingContract: NFT_MANAGER, poolIndex: 0 }),
10    targetPoolId: bytes32(0)
11 });

```

With **A** supplied as a source and **B** as `depositPosition`, **B** passes its exact opt-in and zap binding, while **A** matches **B** on both values checked by `_checkSourceConfig()`. The keeper can supply the swaps needed to convert **A**'s collected fees and rewards into **B**'s pool assets, and the resulting liquidity is added to **B**.

The V4 regression suite establishes the opposite pool-level invariant.

`test_nft_compound_multiple_for_uniswapv4_reverts_when_source_pool_a_targets_pool_b()`

configures two positions under the same manager but in different pools and expects `TargetPoolMismatch`. The equivalent V3 call succeeds because both `PoolIds` have been normalized to zero.

Positions within the same correctly identified pool remain interchangeable if each carries a valid target opt-in. That is consistent with the current pool-level configuration model and is not the issue described here.

Impact

Low. The destination must be an individually owner-enabled NFT, and the operation moves only the selected sources' harvested fees and rewards rather than their principal. Nevertheless, the keeper can move those proceeds into a pool the source configuration cannot distinguish or exclude. This can change the assets and LP exposure in which the proceeds are reinvested — for example, converting fees intended for a WBTC/WETH target into USDC/WETH liquidity — despite the pool-level restriction enforced for equivalent V4 positions.

Recommendation

Carry a V3 pool identity in `MultiCompoundConfig` instead of treating zero as sufficient for every legacy NFT target. For example, add a separate target pool address:

```

1 struct MultiCompoundConfig {
2     bool enabled;
3     Farm target;
4     bytes32 targetPoolId; // V4-style targets
5     address targetPool;   // V3-style targets; zero otherwise
6 }

```

For a V3 destination, `_checkDepositTarget()` should derive the candidate position's actual pool through its connector and require it to equal `config.targetPool`. `_checkSourceConfig()` should then compare each source's configured `targetPool` with the validated destination pool, just as it currently compares `targetPoolId` for V4. Alternatively, replace both fields with one domain-separated pool key that encodes either a V4 `PoolId` or a V3 pool address.

Do not require a destination `tokenId` unless source-to-specific-position routing is an intended product feature. Pool identity is sufficient to restore the invariant already enforced for V4 without introducing stale references when a target NFT is rebalanced and receives a new token ID.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commits 7c517ee8 and f768c5cc.

- The destination declares its pool and every source must agree with it.
- A V3-style destination reports a zero `PoolId`, so leaving `targetPool` unset would make all sources compare equal. Such a config is now rejected at write time with `DestinationPoolNotDeclared`.
- The cleaner long-term fix is using the pool address as the farm identifier. That is a larger change and we have not made it.

2.7.4 No-op deposits leave the ERC-20 compound target unbound from the zap pool

ERC-20 multi-compound settings pin only a `Farm`; the keeper independently controls the zap pool and LP token. With no-op deposit connectors, the action can create a different LP or no LP while reporting the configured target.

Technical Details

The multi-compound feature allows a Sickle owner to opt source positions into automated reward consolidation and pin a target farm into which those rewards should be deposited through `MultiCompoundSettingsRegistry.setErc20Config()`. For ERC-20 targets, `MultiCompoundConfig.target` stores only the `stakingContract` and `poolIndex` fields of a `Farm` struct. When a keeper invokes `MultiFarmStrategy.compoundMultipleFor()`, the strategy validates that each source position's stored target matches `params.depositFarm` by comparing those two fields in `_checkSourceConfig()`. However, the actual destination asset is created earlier by `ZapLib.zapIn()`, which accepts independent keeper-supplied parameters in `params.zap.addLiquidityParams`. This structure includes the liquidity router, pool-specific data, and the resulting LP token address, none of which are validated against the owner-pinned target configuration.

After harvesting rewards and charging fees, `_compoundMultiple()` invokes `ZapLib.zapIn()` with the keeper's zap parameters, then resolves the connector for `params.depositFarm.stakingContract` and calls `deposit(params.depositFarm, params.zap.addLiquidityParams.lpToken, ...)`. The `CurvePoolConnector.deposit()`, `MerkleConnector.deposit()`, and `HydrexMerkleConnector.deposit()` implementations are no-ops because the LP token remains in the Sickle, so they ignore both the `Farm` argument and the token parameter and return successfully without verifying any relationship between the pinned farm and the supplied LP token.

`ZapLib.zapIn()` treats `addLiquidityParams.lpToken == address(0)` as an early successful return after executing swaps, skipping the `addLiquidity` step entirely. A no-op connector's `deposit()` then succeeds without performing any destination operation. In both variants, the final transaction emits events referencing the stored farm, while the actual pool used by the zap may differ or no liquidity may have been added at all.

Impact

Low. A keeper holding the `MULTI_COMPOUND` permission can convert a user's newly claimed rewards into an LP position from a different supported pool than the owner-pinned

target, or complete the operation without adding liquidity by supplying a zero LP token. The resulting assets remain owned by the Sickle and the source principal is not withdrawn, so the operation does not steal funds. The owner can sweep or unwind the assets using standard Sickle controls. The residual harm includes unauthorized market exposure, recovery costs to unwind unwanted positions, misleading compound events that report the pinned farm rather than the actual pool, protocol fees charged despite no target deposit when the zero-LP variant is used, and the potential for one reimbursed ineffective operation per gas-account cadence.

Recommendation

Reject `lpToken == address(0)` in `compoundMultipleFor()` before harvesting to prevent no-deposit variants. Extend `MultiCompoundConfig` for ERC-20 targets to include the canonical destination LP token address and, where applicable, the liquidity router or pool identifier. Before invoking `ZapLib.zapIn()`, validate that `params.zap.addLiquidityParams.lpToken` and related pool fields match the stored target configuration. For connectors that implement no-op deposits, add explicit target validation rather than relying on their `deposit()` call to enforce the relationship between the pinned farm and the supplied token. Emit the validated LP token or pool identifier in compound events so monitoring systems reflect the actual destination used.

Developer Response

Partially fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 8d96cf82.

- A zero destination `lpToken` is rejected before harvesting, rather than charging the fee, running the swaps and depositing nothing.
- The general case stays open: the no-op deposit connectors (Curve pool, Merkl, HydrexMerkl) ignore both farm and token by design, because the LP stays in the Sickle.

2.7.5 NFT claim farm is not bound to the owner's automation opt-in

NFT multi-compound opt-ins bind the NFT key and destination but not the source farm. A keeper can pair an enabled key with a different connector-backed source and trigger reward handling that the owner did not authorize.

Technical Details

The multi-compound automation feature allows Sickle owners to opt individual ERC-20 or NFT positions into keeper-driven reward consolidation. Owners configure each NFT source through `MultiCompoundSettingsRegistry.setNftConfig()` with a key of `(Sickle, nftManager, tokenId)` pinned to a destination farm. An authorized keeper can then trigger `MultiFarmStrategy.nftCompoundMultipleFor()` through `Automation.executeFor()`, which should only harvest from the opted-in sources.

The validation logic in `_validateMultiCompoundFor()` retrieves the stored NFT config using only the keeper-supplied `nftManager` and `tokenId` values, then confirms that the stored config's destination farm matches the requested one. However, this validation never binds the config to the source farm. When `_harvestNftPositions()` executes, it resolves the connector from `claim.position.farm.stakingContract` and forwards `claim.harvest.extraData` directly to that connector. Neither the farm address nor the metadata contents appear in the NFT key consumed through `MultiCompoundSettingsRegistry.getNftConfig()`.

`LineaVestingConnector.claim()` uses `position.farm.stakingContract` as the primary vesting contract address and accepts an optional `secondVestingContract` from the decoded `extraData`. When `hasGauge` is false, the connector calls `releasable(address(this))` and `release()` on both vesting contracts without referencing the validated `nftManager` or `tokenId`. An attacker-controlled keeper can therefore pair an enabled NFT key with either registered Linea vesting address, set `hasGauge = false`, and cause that vesting contract to release all tokens owed to the Sickle, bypassing the owner's per-source opt-in. The structural mismatch means that enabling NFT key A authorizes the keeper to supply NFT key A's identity alongside source farm B's address and claim rewards from B. After the connector releases B's rewards into the Sickle via delegatecall, the same atomic transaction charges the 180-basis-point compound-for fee and executes keeper-selected swaps before depositing into the owner-pinned destination. The operation succeeds because validation never compared the supplied farm to the stored config, and malformed metadata merely causes an atomic revert.

Impact

Low. An authorized keeper with multi-compound permission can release all currently vested rewards from an unconfigured Linea vesting contract by pairing it with an enabled NFT key's identity. The released tokens flow through the compound-for fee calculation and keeper-selected swap routes before reaching the owner-pinned destination farm. A large vesting entitlement can suffer meaningful loss through the fixed fee and slippage, and the owner may face unwanted exposure to the destination farm's lockup or market conditions. The keeper cannot redirect the final deposit to an arbitrary address through this mechanism, but the bypass of source consent allows unauthorized handling and reallocation of rewards the owner did not opt into multi-compounding.

Recommendation

Add a `Farm` field to the stored `MultiCompoundConfig` in `MultiCompoundSettingsRegistry` and require the owner to record the source farm alongside the destination when calling `setNftConfig()`. Modify `_validateMultiCompoundFor()` to compare the keeper-supplied `claim.position.farm` against the stored source farm before resolving any connector. For the Linea integration, remove `secondVestingContract` from individual claim metadata and require each vesting contract to have its own enabled NFT config pinned to the same destination, or extend the config to store and validate an allowed secondary contract address. Alternatively, introduce a connector validation hook that returns a canonical list of all source contracts affected by the claim metadata and require each returned source to have an enabled config before executing any Sickle delegatecall.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commits 7c517ee8 and 11a20555.

- NFT sources bind to the farm recorded in their opt-in; a mismatch reverts with `SourceFarmMismatch`.
- Self-target configs must declare the source farm, enforced at write time.

Additional Linea hardening was completed in <https://github.com/vfat-io/sickle-contracts/pull/917>, commits 7767bc44, bdf5fa0a, and cb26a02f (merge 28df5d08).

Both vesting contracts are constructor-pinned, the payload no longer supplies a vesting address, and any optional counterpart is derived from those immutables. `claim()` and `earned()` reject unpinned primary contracts. Connector redeployment is still required for this hardening to become live.

2.7.6 Partial rebalance can orphan the original NFT after settings migration

`rebalanceFor()` does not require full removal of the old position before migrating settings. A keeper can leave most liquidity on the original NFT while all policies move to a small replacement.

Technical Details

`NftFarmStrategy.rebalanceFor()` accepts a keeper-supplied `NftRebalance` struct that includes a `withdraw.zap.removeLiquidityParams` field specifying the manager, token ID, and liquidity amount to remove from the old position. The function reaches `_rebalance()`, which calls `_withdraw()`. That helper forwards these parameters directly to `NftZapLib.zapOut()` and then to the liquidity connector's `removeLiquidity()` method.

The in-scope `UniswapV3Connector.removeLiquidity()`, `UniswapV4Connector.removeLiquidity()`, and `PancakeInfinityConnector.removeLiquidity()` functions implement partial liquidity removal: when the requested `liquidity` is less than the position's current liquidity, they decrease the position but do not burn the NFT. For example, `UniswapV3Connector.removeLiquidity()` burns the NFT only when the remaining liquidity reaches zero after the decrease operation. After `_withdraw()` completes, `_rebalance()` calls `_resetNftSettings()`, which invokes `NftSettingsRegistry.transferNftSettings()`. That function queries the connector for the latest token ID owned by the Sickle, assumes it represents a new replacement, and unconditionally deletes all core settings for the old `tokenId` while installing them on the new token. It then iterates over each registered settings module, such as `MultiCompoundSettingsRegistry.onRebalance()`, and migrates every module policy from the old key to the new key.

The pre-mint checks in `ShiftMath.preMintRebalanceChecks()` and the post-mint checks in `postMintRebalanceChecks()` verify that the replacement NFT is minted to the correct pool and, in Shift mode, to the formula-determined tick range. Neither check validates that the old NFT was burned, that all of its liquidity was removed, or that the keeper-supplied `removeLiquidityParams.nft` and `tokenId` match the configured position key. As a result, a keeper can specify a negligible removal amount, leave the old NFT live and liquid in the Sickle, and commit a state transition in which all automation policies follow a smaller replacement while the economically dominant old NFT becomes unstaked and policy-less.

Impact

Low. A malicious or compromised keeper can use a partial liquidity removal to leave most of the user's position in the original, still-live NFT while migrating all core settings and module policies to a newly minted replacement with minimal liquidity. The old NFT is withdrawn from its farm, loses its auto-exit, reward, rebalance, multi-compound, and stake configurations, and no longer receives farm rewards or automated risk management. The user's assets remain under Sickle custody, but normal automation follows only the small replacement.

A regular Sickle owner may manually recover or reconfigure the old NFT, though the standard position metadata still references the farm from which the malicious rebalance already withdrew

it. For vault-backed Sickles with a sensibly strict `maxRebalanceLossBps`, the vault callback should revert when the orphaned value is excluded from replacement accounting, mitigating the issue in that deployment path. Vaults configured with a 100% rebalance-loss tolerance can commit the state transition and lose visibility of the orphaned NFT. The primary harm is lost farming, automation continuity, and recovery burden rather than theft.

Recommendation

In `rebalanceFor()`, require that `params.withdraw.zap.removeLiquidityParams.nft` equals `params.position.nft` and `params.withdraw.zap.removeLiquidityParams.tokenId` equals `params.position.tokenId` before calling `_withdraw()`. Require `params.withdraw.zap.removeLiquidityParams.liquidity` to be set to the full-removal sentinel `type(uint128).max`. After `zapOut()` completes, verify through the connector that the old token no longer exists or that its remaining liquidity is exactly zero before minting the replacement or calling `_resetNftSettings()`. Pass the strategy-discovered replacement token ID explicitly into the registry migration function and have the registry reject any populated destination key, rather than independently assuming that `getTokenId()` represents a collision-free one-for-one transition. Extend post-mint checks to assert that the old position is closed, the new position is owned by or validly staked for the same Sickle, and the new pool and range are correct.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](#), commit 6f23682a.

- Automated rebalance must fully drain the source position; a partial removal reverts with `RebalancePartialRemoval`.
- We compare against live liquidity rather than restoring `nftSupplyUnchanged`: V4 and Pancake Infinity implement `totalSupply()` as `nextTokenId() - 1`, a mint counter that never decrements, so a supply check would revert every V4 rebalance while passing on V3.

2.7.7 Automated compound can override the configured unstaked custody mode

`compoundFor()` lets the keeper choose `inPlace`. On an unstaked Slipstream position, `false` turns an idempotent no-op withdrawal into an unconditional redeposit, bypassing the separate stake-mode controller.

Technical Details

The `NftFarmStrategy.compoundFor()` function accepts a keeper-controlled `inPlace` parameter that determines whether an NFT position will be withdrawn before zapping and redeposited afterward. When `inPlace` is false, `_compound()` calls `_withdrawNft()` before the zap and `_depositNft()` afterward, without checking whether the NFT was staked before the operation or verifying that the final custody state matches the initial state.

This design conflicts with the `StakeStrategy` module, which enforces custody-mode transitions based on on-chain yield ratio signals, threshold checks, and configured dwell periods. The `SlipstreamGaugeConnector.withdrawNft()` implementation is intentionally idempotent: when an NFT is not staked in the gauge, the connector checks `stakedContains()` and skips the withdrawal rather than reverting. When the keeper calls `compoundFor()` with `inPlace=false` while the NFT is Sickle-custodied (unstaked), the pre-zap withdrawal succeeds silently without

moving the NFT, the zap modifies liquidity on the Sickle-held position, and `_depositNft()` unconditionally deposits the NFT into the gauge. The operation completes successfully but changes custody mode rather than preserving it, bypassing all ratio, threshold, and dwell enforcement in `StakeStrategy.stakeFor()`.

The call requires that the Sickle holds usable pool-token balances to produce nonzero added liquidity. Residual fees or dust from previous operations can satisfy this requirement. Existing validation in `NftSettingsRegistry.validateCompoundFor()` and `ShiftMath.checkCompoundTargetBindings()` protects NFT manager and token ID binding but does not constrain the keeper-supplied `inPlace` value or enforce custody-mode consistency.

Impact

Low. This outcome does not require a malicious keeper or deliberately malformed parameters. Slipstream compounds normally use `inPlace = false` while the position is gauge-staked. If `StakeStrategy` subsequently performs a legitimate transition to the unstaked mode, an ordinary compound that reuses this configured value observes an idempotent no-op withdrawal and then unconditionally deposits the NFT back into the gauge. The compound therefore reverses the controller's on-chain decision without satisfying its ratio, threshold, or dwell conditions.

Vault shareholders may lose the incremental swap-fee yield expected during the intended unstaked interval and incur unnecessary gas from alternating custody transitions. Principal remains withdrawable, and a later qualifying `unstakeFor()` sequence can restore the intended mode, although its signal and dwell period must be established again. Routine compounds can repeatedly shorten unstaked intervals until the execution path derives `inPlace` from current custody. The harm is therefore bounded and recoverable but can systematically undermine the configured yield-optimization policy, supporting Low severity.

Recommendation

Query the farm connector's `isStaked(address(sickle), position)` before executing the compound operation. Derive the execution shape from the current custody state rather than accepting the keeper-provided `inPlace` parameter: if the NFT is unstaked, perform the zap in place and do not call `depositExistingNft()`; if it is staked and the connector requires withdrawal, withdraw and restore the same staked state. After the operation, query `isStaked()` again and assert that the result matches the pre-call snapshot. Any mismatch should revert the entire claim, zap, fee charge, and transfer sequence. This approach preserves custody mode as an invariant and prevents the keeper from overriding the owner's configured stake controller policy.

```

1 function _compound(
2     Sickle sickle,
3     NftPosition calldata position,
4     NftCompound calldata params,
5     bool inPlace,
6     address[] calldata sweepTokens,
7     bytes4 fee
8 ) private {
9     + // Snapshot custody state before compounding
10    + address connector = connectorRegistry.connectorOf(position.farm.
11      stakingContract);
12    + bool wasStaked = INftFarmConnector(connector).isStaked(address(sickle),
13      position);
14    +
15    _claim(sickle, position, params.harvest, fee);

```

```
15 -   if (!inPlace) {
16 +   // Withdraw only if the NFT was initially staked
17 +   if (wasStaked) {
18       _withdrawNft(sickle, position, params.harvest.extraData);
19   }

21   _zapIn(sickle, params.zap);

23 -   if (!inPlace) {
24 +   // Redeposit only if the NFT was initially staked
25 +   if (wasStaked) {
26       _depositNft(sickle, position, params.harvest.extraData);
27   }
28 +
29 +   // Assert custody state is preserved
30 +   bool isStaked = INftFarmConnector(connector).isStaked(address(sickle),
31   position);
31 +   if (isStaked != wasStaked) revert CustodyModeChanged();

33   _sweep(sickle, sweepTokens);

35   emit SickleCompoundedNft(
36       sickle,
37       position.nft,
38       position.tokenId,
39       position.farm.stakingContract,
40       position.farm.poolIndex
41   );
42 }
```

If some connectors cannot reliably expose custody state, store an owner-selected custody policy in the settings registry and enforce it per integration rather than trusting an unauthenticated keeper boolean.

Developer Response

Acknowledged as part of the keeper trust mode.

2.7.8 Generic `executeFor()` rebalances skip vault safeguards

`Automation.executeFor()` can dispatch `NftFarmStrategy.rebalanceFor()` without the vault callbacks used by the typed rebalance entrypoint. If this route is used for a vault after automatic rebalancing is enabled, an otherwise valid keeper operation can leave the vault referencing the retired NFT and bypass its rebalance loss limit.

Technical Details

`Automation.rebalanceFor()` calls `NftFarmVault.beforeRebalance()` before strategy execution and `NftFarmVault.onRebalance()` afterward. These callbacks snapshot the old position value, validate the replacement's value and custody, enforce `maxRebalanceLossBps`, and update `position.tokenId`. In contrast, `_executeOne()` validates the generic action and then calls the strategy directly with `strategy.call(data)`.

The `NftFarmStrategy.rebalanceFor()` selector is explicitly registered for generic `executeFor()` dispatch under `automationActions.NftFarmStrategy` in both the [Base canary deployment](#) and the [Linea canary deployment](#). Vault Sickles also consent to `NFT_REBALANCE` during construction. The vault deployment helper currently initializes automatic rebalancing as disabled, so reaching the faulty path additionally requires the vault admin to enable it and an authorized keeper to select the registered generic route.

`NftFarmStrategy.rebalanceFor()` can then retire the old NFT, mint and deposit its replacement, and migrate its settings while neither vault callback runs. The vault continues storing the old token ID because only `NftFarmVault.onRebalance()` records the replacement. Subsequent deposits and redemptions resolve the retired position and revert, while the configured rebalance value floor was never enforced.

Impact

Low. The route is restricted to an authorized keeper and current vault deployment settings leave automatic rebalancing disabled. If enabled and the registered generic route is used, one valid rebalance can block normal vault deposits and redemptions and skip the vault's loss protection.

Recommendation

Remove `NftFarmStrategy.rebalanceFor()` from `automationActions.NftFarmStrategy` in the Base and Linea canary deployment plans and reject callback-dependent selectors in `executeFor()`. If generic dispatch must support vault rebalances, register immutable pre-call and post-call hooks with the action and execute both around the strategy call.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](#), commits [ba265285](#) and [d9f6696c](#).

- `rebalanceFor` dropped from generic `executeFor()` dispatch; it is reachable only through the typed entrypoint, which runs the vault checks.
- Both canaries updated on-chain: `actionPermission` for `0x4db46eab` is now 0. `harvestFor` / `exitFor` / `compoundFor` unchanged at 8/32/16.

2.7.9 Gas-account mode can be disabled after execution but before reimbursement

`Automation._executeOne()` checks gas-account mode before executing a strategy, but `Automation._chargeGasAccount()` checks the mutable mode again before reimbursing the keeper. A contract-based Sickle owner can disable gas-account mode from a callback during strategy execution, causing the completed operation to skip reimbursement and cadence recording.

Technical Details

`_executeOne()` calls `_preflightGasAccountCadence()` before dispatching the strategy. The preflight considers a nonzero `GasAccount.maxChargeableFee` to mean gas-account mode and validates the charge cadence. `_executeOne()` then makes an untrusted external call to the selected strategy before calling `_chargeGasAccount()`.

Strategies commonly finish by sweeping assets to the Sickle owner through `TransferLib.transferTokensToUser()`. A native-token transfer gives a contract-based owner a callback. From that callback, the owner can call `GasAccount.setMaxChargeableFee(sickle, 0)`, since the setter authorizes the Sickle owner. Execution then returns to `_chargeGasAccount()`, which reads `maxChargeableFee` again and returns immediately because the cap is now zero. The strategy's state changes persist, but the keeper receives no reimbursement and the prepaid balance is not debited. For cadence-gated permissions, `GasAccount.charge()` is never reached, so `lastChargedAt` is not updated either. A reentrancy guard on `Automation` would not prevent this sequence because the callback modifies `GasAccount` directly.

Impact

Low. A malicious contract-based owner can receive the benefit of an automation operation while making the keeper pay its gas cost. The owner can also avoid recording the charged-operation cadence, although keepers can mitigate repeated abuse through simulation and blacklisting.

Recommendation

Commit gas-account mode before entering the strategy and make that commitment immutable until settlement. A robust design is a two-phase `GasAccount` flow:

1. `beginCharge` validates cadence, snapshots the cap, and reserves sufficient balance or marks the Sickle as having an active charge.
2. Owner configuration and withdrawals cannot affect the committed operation.
3. `settleCharge` pays the measured amount, refunds any unused reservation, records cadence, and clears the lock.

At minimum, preserve the initial mode and force post-call settlement against a committed cap rather than re-reading `maxChargeableFee` and silently switching to fee mode. Rejecting cap changes while a charge is active would also close the callback window.

Developer Response

Fixed in [vfat-io/sickle-contracts#895](https://github.com/vfat-io/sickle-contracts/pull/895), landed on `main` via #891, commit b21233d3.

- Gas-account mode is committed before the strategy runs, so disabling the cap mid-execution cannot skip reimbursement.

2.7.10 Rollup fee oracles are queried with inner strategy calldata rather than the priced transaction

`Automation._chargeGasAccount()` passes one strategy payload to the gas-cost adapter, but the OP Stack and Scroll fee oracles price the data of the outer transaction. Gas-account-mode Sickles can therefore reimburse a keeper for a data fee that does not match the fee actually charged by the rollup.

Technical Details

`executeFor()` receives the strategy payload as one element of the outer `callData` array. `_executeOne()` forwards that isolated element as `data` to `_chargeGasAccount()`, which calls `IGasCostAdapter.txCost(gasUsed, data)`. The supplied bytes contain only the strategy selector and arguments. They omit the `executeFor()` selector, ABI array offsets and lengths, the other transaction arguments, and any other batch items.

`OpStackGasCostAdapter._l1Fee()` supplies these inner bytes to `getL1Fee()`, and `ScrollGasCostAdapter._l1Fee()` does the same. The [OP Stack documentation](#) specifies that `getL1Fee()` takes a serialized unsigned Ethereum transaction. Likewise, the [Scroll documentation](#) specifies a signed, fully RLP-encoded transaction and explains that the fee depends on the transaction data's size and byte composition. The isolated strategy payload satisfies neither interface expectation.

Pricing a differently encoded inner call can either understate or overstate the keeper's actual data cost. The mismatch exists even when `executeFor()` contains only one operation and is separate from duplicating transaction-wide fees across a batch.

Impact

Low. Keepers can be systematically underpaid or overpaid on OP Stack and Scroll deployments. Underpayment can make otherwise valid automation uneconomic, while overpayment depletes users' capped prepaid gas balances faster than the transaction's actual cost.

Recommendation

Separate per-operation execution reimbursement from the transaction-wide rollup data fee. Calculate the data fee once for the complete `executeFor()` transaction and allocate that single amount across the batch using a documented deterministic rule, such as each operation's proportional contribution to the outer ABI-encoded calldata. Charge each Sickle its measured execution cost plus only its allocated share, and enforce that the aggregate data-fee allocation does not exceed the single batch-level fee.

Because the executing contract cannot reconstruct the signed RLP transaction faithfully, the approved automator can calculate the batch fee and per-Sickle allocations off-chain and submit them with the transaction. This is consistent with the automator's trusted role, while each Sickle's configured maximum charge and cadence remain on-chain limits on the submitted reimbursement. If this trust assumption is undesirable, reimburse the shared data fee once from protocol funds or restrict gas-account batches to a single Sickle.

Developer Response

Not fixable at this layer. Documented in [vfat-io/sickle-contracts#894](#), commit d4701c3b.

2.7.11 Batched gas-account reimbursements duplicate transaction-wide fixed fees

`Automation.executeFor()` charges each item in a batch independently, but some gas-cost adapters include components that are incurred only once by the outer transaction. On affected chains, a keeper can therefore receive repeated reimbursement for one fixed cost and progressively deplete opted-in Sickles' prepaid gas balances.

Technical Details

`executeFor()` invokes `_executeOne()` for every batch item. After the strategy call, `_executeOne()` measures that item's strategy gas and calls `_chargeGasAccount()`, which invokes `IGasCostAdapter.txCost()` and immediately charges the resulting amount from the Sickle's `GasAccount`. The adapter receives each item separately, so the calculation has no context for fixed costs already shared by the enclosing transaction.

Some adapter components are fixed once per outer blockchain transaction:

- `OpStackGasCostAdapter._operatorFee()` adds `operatorFeeConstant` to every item;
- `EtherlinkGasCostAdapter.txCost()` adds `inclusionFixedBytes × inclusionFeePerByte` to every item.

For a batch of `N` operations, the chain charges one transaction-wide fixed component, while the keeper receives it `N` times from users' gas accounts. The code comment in `_executeOne()` explicitly states that transaction-wide fixed costs must not be charged per item, but the adapters do exactly that.

Independent calls to an L1 fee oracle with each inner payload can introduce further duplicated envelope overhead, although the explicit constants alone establish the overcharge.

Impact

Low. Batched automation can systematically overcharge gas-account-mode Sickles and exhaust prepaid balances faster than the keeper's actual transaction cost, particularly when many lightweight operations are included in one transaction.

Recommendation

Calculate transaction-wide fixed components once in `executeFor()` and allocate them across the batch, or make adapters return only genuinely marginal per-operation costs. Data-fee and envelope calculations should likewise account for the outer transaction once before distributing the resulting cost among the operations.

Developer Response

Fixed in [vfat-io/sickle-contracts#895](https://github.com/vfat-io/sickle-contracts/pull/895), landed on `main` via #891, commit b21233d3.

- `IGasCostAdapter.txCost()` takes `batchSize` and shares the transaction-wide components across the batch.
- ABI break: selector `0x302bf43f` → `0x8a563399`. `Automation` and the adapter must be deployed together.

2.7.12 Compound targets remain usable after an NFT leaves Sickle custody

`compoundFor()` validates persisted settings and the target NFT identity, but it does not verify that the NFT is still owned by the Sickle or remains staked on its behalf. If the NFT leaves Sickle custody while its settings remain enabled, an approved automator can direct Sickle-held rewards or idle balances into the stale position.

Technical Details

`NftSettingsRegistry.validateCompoundFor()` checks only the `automateRewards` flag and compound behavior stored under the `(sickle, nftManager, tokenId)` key.

`ShiftMath.checkCompoundTargetBindings()` additionally requires `addLiquidityParams.nft` and `addLiquidityParams.tokenId` to match the supplied position. These checks establish calldata identity, but do not query `ownerOf()` or connector-specific staking state to establish current custody.

`simpleWithdraw()` and `simpleExit()` transfer the NFT out through `_transferOutNft()` without clearing its settings. An approved keeper can subsequently call `compoundFor()` with `inPlace = true`, which skips the farm withdrawal and redeposit steps and invokes `_zapIn()` directly. The zap consumes balances held by the Sickle and the selected connector increases liquidity on the persisted manager and token ID. For connectors that permit or otherwise authorize this increase, the funds are added to an NFT controlled by its new owner.

Impact

Low. An approved automator can cause Sickle-owned rewards or idle assets to be irreversibly added to an NFT that is no longer controlled by the Sickle. The issue is limited to configured compound targets and does not directly transfer the NFT principal.

Recommendation

- Clear automation settings when an NFT leaves Sickle custody, or
- Require current Sickle ownership or recognized Sickle staking before compounding.

The check should validate custody immediately before the add-liquidity action, not merely when settings are first stored.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 2c9f02dd.

- Custody check in `ShiftMath.checkCompoundTargetBindings()`, which `compoundFor()` already calls, so it costs the strategy two extra arguments.
- Follow-up in #899 (202af112) and #901 (fea13fda, 191e88c4) — see #2 for the connector-level detail.
- We chose this over clearing settings on transfer-out: the registry's unset does not cascade to settings modules, so clearing would have left the MultiCompound equivalent untouched and needed a new Sickle-callable unset on a contract deployed to 51 chains.

2.7.13 Automated rebalances can remove liquidity from an unauthorized NFT

`rebalanceFor()` validates automation settings for one NFT but allows the keeper to select a different NFT as the liquidity-removal source. An authorized automator can therefore alter or burn a Sickle-owned position for which automatic rebalancing was not enabled.

Technical Details

`NftFarmStrategy.rebalanceFor()` constructs the validated `NftKey` from `params.position.nft` and `params.position.tokenId`. The preflight helper `ShiftMath._checkRebalanceTargetBindingsInner()` validates the replacement's `addLiquidityParams`, including its manager and fresh-mint token ID, but does not validate `params.withdraw.zap.removeLiquidityParams` against the same key. `_rebalance()` then passes this independent withdrawal structure to `_withdraw()`. `NftZapLib.zapOut()` selects a connector from `removeLiquidityParams.nft` and calls `removeLiquidity()` with the independently supplied `removeLiquidityParams.tokenId`.

Neither field is required to match the NFT manager and token ID in the validated key. A keeper can make validation succeed with an out-of-range position A, remove liquidity from another Sickle-owned position B, and use the proceeds to mint position C in A's configured pool. The post-mint checks only validate C's pool identity and ticks, so the transaction can succeed while B is altered or burned and A's settings are migrated to C.

Impact

Low. An authorized automator can bypass per-position rebalance consent and partially or fully remove liquidity from another Sickle-owned NFT.

Recommendation

Extend `ShiftMath._checkRebalanceTargetBindingsInner()` with the removal-source binding next to the existing add-liquidity target checks. Require `params.withdraw.zap.removeLiquidityParams.nft` to equal `key.nftManager` and its `tokenId` to equal `key.tokenId` before `_rebalance()` executes.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 0f712546. Same binding as #4.

2.7.14 Automated exits and rebalances do not bind the removal source to the authorized NFT

Technical Details

`NftFarmStrategy.exitFor()` validates the automation settings for the supplied `position`, then forwards independently supplied withdrawal parameters.

`NftFarmStrategy._exit()` eventually calls `NftFarmStrategy._withdraw()`, which withdraws the validated position from its farm but passes the unrelated `withdrawParams` to `NftFarmStrategy._zapOut()`.

`NftZapLib.zapOut()` selects the connector and removes liquidity according to `params.zap.removeLiquidityParams.nft` and `tokenId`. Neither value is compared with the position that passed the settings check.

`NftFarmStrategy.rebalanceFor()` has the same problem. Its pre-mint checks bind the requested new target and range, but do not require the remove-liquidity source in `params.withdraw` to equal `params.position`.

Thus, permission to automate position **A** can be used to remove liquidity from another NFT **B** held by the same Sickle, even if **B** has no corresponding exit or rebalance consent.

Impact

Low. An authorized keeper can forcibly remove or reallocate liquidity belonging to another Sickle-owned NFT. This can:

- close or materially alter a position for which automation was not enabled;
- realize fees and losses at an unintended time;
- execute caller-selected swaps and incur slippage;
- leave the authorized position withdrawn but otherwise unchanged.

Recommendation

Before automated exit or rebalance execution, require:

```
1 NftRemoveLiquidity calldata remove =
2     withdrawParams.zap.removeLiquidityParams;

4 if (
5     remove.nft != position.nft
6     || remove.tokenId != position.tokenId
7 ) {
8     revert WithdrawPositionMismatch();
9 }
```

Developer Response

Fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit 0f712546.

- Rebalance binds the removal source in `ShiftMath._checkRebalanceTargetBindingsInner()`, alongside the existing target checks, so the strategy pays no bytecode.
- `exitFor()` binds inline — there is no `ShiftMath` call on that path, and adding one for two comparisons costs an external call on every automated exit.
- Added regression tests for both.

2.7.15 The Arbitrum gas adapter cannot accurately price the variable L1 poster fee

Technical Details

`Automation._executeOne()` measures gas only around the inner strategy call:

```
1 // contracts/Automation.sol
2 uint256 gasBefore = gasleft();
3 (bool success, bytes memory result) =
4     strategy.call{ value: value }(data);
5 uint256 gasUsed = gasBefore - gasleft();

7 // ...
8 _chargeGasAccount(sickle, permission, gasUsed, data);
```

`ArbitrumGasCostAdapter.txCost()` then charges only that measured gas plus a fixed overhead:

```
1 // contracts/GasCostAdapter.sol
2 function txCost(
3     uint256 l2GasUsed,
4     bytes calldata
5 ) external view returns (uint256) {
6     return _executionCost(l2GasUsed);
7 }
```

On Arbitrum Nitro, the L1 poster cost is converted to L2 gas units and deducted from the transaction's available gas before EVM execution starts. It is therefore included in transaction-level gas accounting, but not in a `gasleft()` delta measured around an inner contract call. This behavior is visible in Nitro's [transaction processor](#), where `posterGas` is calculated and charged before the EVM is entered.

The adapter also ignores its `bytes calldata` argument. A sufficiently large fixed `overhead` may partially or fully compensate for the omitted poster fee under particular conditions, but it cannot track changes in calldata compressibility or the L1 base fee and therefore cannot guarantee an accurate reimbursement.

Impact

Low. The adapter does not necessarily under-reimburse every operation: the configured `overhead` may offset or exceed the omitted costs. When it is lower than the unmeasured execution costs plus the poster fee, the keeper is under-reimbursed and may stop executing economically unviable automation. When it is higher, the Sickle is overcharged. Because the poster fee varies with calldata compressibility and L1 pricing while `overhead` is fixed, the reimbursement cannot consistently match the keeper's actual cost.

Recommendation

Keep `overhead` for the fixed or marginal execution costs outside the metering window, but account for the variable poster fee separately using the supported Arbitrum gas precompile, such as `ArbGasInfo`, or redesign settlement so that transaction-wide costs can be reconciled after execution. Do not calibrate a fixed `overhead` as a substitute for the poster fee.

For batched automation, charge the poster fee once and allocate it across the operations; querying and charging the entire transaction fee once per item would overcharge batches. Add an Arbitrum fork test that compares total keeper cost with the total amount reimbursed.

Developer Response

Not fixable at this layer. Documented in [vfat-io/sickle-contracts#894](#), commit d4701c3b.

- The executing contract cannot reconstruct the signed RLP it would need to price the transaction.
- `docs/AutomationKeeperTrust.md` records the gap and what closing it requires: the keeper computes the transaction-level figure off-chain and submits it, bounded on-chain.

2.7.16 A stale multi-compound target can receive rewards after ownership changes

Technical Details

`MultiFarmStrategy._checkDepositTarget()` verifies that the target configuration is enabled and that its farm, pool and add-liquidity target match the supplied `depositPosition`. It does not verify that the NFT is still owned by the Sickle or its owner, or staked on behalf of the Sickle.

The configuration is keyed by `(sickle, nftManager, tokenId)` and persists when the NFT leaves the Sickle. For example, `NftFarmStrategy.simpleWithdraw()` withdraws and transfers the NFT to the owner without clearing the multi-compound configuration. The NFT may also subsequently be transferred outside the protocol.

Only the Sickle owner can call `setNftConfig()` or `unsetNftConfig()`. Therefore, this scenario requires the user to transfer the target NFT without first removing its automation configuration; the keeper cannot independently configure an unrelated target.

When `compoundInPlace` is true, `MultiFarmStrategy._nftCompoundMultiple()` skips both `MultiFarmStrategy._withdrawNft()` and `MultiFarmStrategy._depositNft()` and directly calls `NftZapLib.zapIn()`. Adding liquidity to an existing position is normally permissionless, so this call can still succeed after the NFT has been transferred to a different owner.

Impact

Low. Rewards harvested from several Sickle positions can be irreversibly donated to the current owner of an NFT that no longer belongs to the Sickle. The loss can include all rewards selected by an authorized keeper for the multi-compound operation. The loss is conditional on the Sickle owner leaving the configuration enabled after transferring the NFT.

Recommendation

Add the following error at contract level:

```
1 error DepositPositionNotControlled();
```

Then insert this check in `MultiFarmStrategy._checkDepositTarget()`, after binding the zap target to `depositPosition`:

```
1 address nftOwner =
2     depositPosition.nft.ownerOf(depositPosition.tokenId);
3
4 if (
5     nftOwner != address(sickle)
6     && nftOwner != sickle.owner()
7 ) {
8     address farmConnector = connectorRegistry.connectorOf(
9         depositPosition.farm.stakingContract
10    );
11
12    if (
13        !INftFarmConnector(farmConnector).isStaked(
14            address(sickle), depositPosition
15        )
16    ) {
17        revert DepositPositionNotControlled();
18    }
19 }
```

This permits a target held by the Sickle, held directly by `sickle.owner()`, or reported by the farm connector as staked for the Sickle.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](#), commit 06f95f90.

- `MultiFarmStrategy._checkDepositTarget()` requires current custody: the Sickle holds the NFT, its owner holds it, or the farm reports it staked for that Sickle.
- Follow-up in #899 (202af112) and #901 (fea13fda, 191e88c4): four connectors answered "staked by anyone" rather than binding the caller. Sushi now uses `getStaker`; Thena/Atlantis/Blackhole and Ultrasolid use `nft.ownerOf`.
- Hybra's gauge publishes no `tokenId`→`depositor` record so it stays token-level. A stale target there is rejected by the gauge's own withdraw authorisation instead.
- Added regression tests, including one that fails against the previous connector behaviour.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Action registrations do not enforce their single-permission-bit invariant

`Automation` documents every `(strategy, selector)` registration as mapping an action to one permission bit, and the deployment tooling only produces such single-bit values. However, `setActionPermission()` accepts any non-zero `uint256`, including a composite mask. Authorization handles that composite as a set of required bits, while `GasAccount` treats it as one exact permission key. A malformed admin registration can therefore have different semantics across the two components.

Technical Details

`actionPermission` and `setActionPermission()` both describe the registered value as a permission bit. The setter nevertheless stores any value without validating that it is zero or a power of two:

```
1 function setActionPermission(
2     address strategy,
3     bytes4 selector,
4     uint256 permission
5 ) external onlyAdmin {
6     actionPermission[strategy][selector] = permission;
7     emit ActionPermissionSet(strategy, selector, permission);
8 }
```

Zero has an explicit meaning: `_executeOne()` treats it as an unregistered action. Every other value is accepted. If an admin registers an action under a composite such as `MULTI_COMPOUND | FARM_HARVEST`, `_requireAuthorized()` requires the owner and automator to hold both bits:

```
1 if (actionConsent[sickle] & action != action) {
2     revert ActionNotConsented(sickle, action);
3 }
4 // ...
```

```

5 if ((globalAutomatorPermissions[msg.sender] & action) != action) {
6     revert NotApprovedAutomator();
7 }

```

The same composite is then forwarded unchanged to `GasAccount`. Unlike authorization, `GasAccount._validateCadence()` and `_recordCadence()` look up the whole value as one mapping key. Consequently, `cadenceRequired[MULTI_COMPOUND]` and `cadenceRequired[FARM_HARVEST]` may both be true while `cadenceRequired[MULTI_COMPOUND | FARM_HARVEST]` remains false. The composite action will not inherit either constituent bit's cadence.

The comment that permission masks are free-form in `AutomationPermissions` applies to grant masks and permits future categories to use higher bits. It does not make composite action registrations part of the configured design: the same library says that each generic-dispatch pair is bound to a bit, and `_automationActionSpec()` resolves every supported action to exactly one named, one-hot permission.

The audited deployment plans therefore do not expose this state. It can only be introduced through a direct or future admin-controlled registration that violates the invariant already followed by the protocol tooling. The concern is configuration hardening: the contract accepts a representation that its documentation, tooling and exact-key cadence accounting do not consistently support. If an admin or future deployment script supplies a composite, cadence limits and monitoring configured for its constituent categories will not apply to that action.

Impact

Informational.

Recommendation

Enforce the documented representation at the boundary. Preserve zero as the unregister value and reject every non-zero value that is not a power of two:

```

1 error InvalidActionPermission(uint256 permission);

3 function setActionPermission(
4     address strategy,
5     bytes4 selector,
6     uint256 permission
7 ) external onlyAdmin {
8     if (permission != 0 && permission & (permission - 1) != 0) {
9         revert InvalidActionPermission(permission);
10    }
11    actionPermission[strategy][selector] = permission;
12    emit ActionPermissionSet(strategy, selector, permission);
13 }

```

Developer Response

Fixed in [vfat-io/sickle-contracts#895](https://github.com/vfat-io/sickle-contracts/pull/895), landed on `main` via `#891`, commit `b21233d3`.

- `setActionPermission()` rejects composite masks; the permission must be a single bit. Zero still means unregistered.

2.9.2 Owner-configured rebalance loss limits are not enforced

`NftSettingsRegistry` stores owner-selected slippage, price-impact, and dust limits for automated rebalances, but `rebalanceFor()` never consumes them. These settings therefore do not constrain the keeper-supplied swap and liquidity parameters used during execution.

Technical Details

The owner-facing `RebalanceConfig` contains `slippageBP`, `priceImpactBP`, and `dustBP`. When the owner enables automated rebalancing, `NftSettingsRegistry._checkRebalanceConfig()` only checks that these values fall within protocol-wide ranges before storing them. This proves that the configuration is syntactically valid, but it does not apply the selected limits to a rebalance.

At execution time, `NftSettingsRegistry.validateRebalanceFor()` verifies that automated rebalancing is enabled, that the current tick has crossed the configured trigger, and that it remains within the cutoff range. It does not use `slippageBP`, `priceImpactBP`, or `dustBP`. `ShiftMath.preMintRebalanceChecks()` loads the settings, but uses them only to bind the destination manager and pool and, when Shift mode is enabled, to derive the expected tick range. The corresponding `postMintRebalanceChecks()` verifies the replacement pool and ticks without checking the value retained in the new position.

After those preliminary checks, `NftFarmStrategy.rebalanceFor()` forwards keeper-supplied parameters to `_rebalance()`. The resulting zap accepts keeper-selected swap routes, `minAmountOut` values, mint minimums, and opaque router calldata. In particular, `AggregatorConnector.swapExactTokensForTokens()` only compares the amount returned to the Sickle against the `minAmountOut` supplied in the same keeper calldata. A zero minimum therefore does not enforce the owner's stored percentage limits, and the opaque router call may specify a different output recipient. A replacement NFT can still pass the pool and tick checks while containing substantially less value than the original position.

An authorized keeper is trusted under the stated threat model to construct conservative, policy-compatible execution parameters. Deliberately choosing a hostile recipient or ineffective minimum is therefore not a privilege escalation by itself. The logical issue is that the protocol exposes these three fields as owner-selected rebalance allowances, validates and migrates them with the position, but never makes them an on-chain execution constraint. Their effective enforcement depends entirely on off-chain keeper behavior.

Impact

Informational.

Recommendation

Enforce the existing owner-configured `slippageBP`, `priceImpactBP`, and `dustBP` limits stored in `NftSettingsRegistry` during automated rebalances. Derive swap and liquidity minimums from these values using a manipulation-resistant quote, require every swap output recipient to be the executing Sickle, and reject rebalances whose resulting value loss or residual dust exceeds the configured bounds.

Do not rely on keeper-supplied `minAmountOut`, desired amounts, or opaque recipient calldata as proof that the owner-selected policy was respected. If reliable on-chain enforcement is not available and these fields are intended to remain advisory, remove them from the owner-facing configuration or clearly document that they do not constrain execution.

Developer Response

Documented in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit d4701c3b.

- Loss limits are advisory keeper inputs, not on-chain constraints. `docs/AutomationKeeperTrust.md` says so explicitly.

2.9.3 Equalizer NFPG registry accepts non-canonical targets with copied metadata

`EqualizerNfpgGaugeRegistry` identifies an NFPG from values reported by the candidate contract itself. A noncanonical target can copy the metadata of a genuine NFPG and be assigned `EqualizerNfpgConnector`, even though no canonical factory or voter relationship authenticates the target address.

Technical Details

`EqualizerNfpgGaugeRegistry.connectorOf()` probes the supplied `target` for `pool()`, `token0()`, `token1()`, `tickSpacing()`, and `gauge()`. It then verifies that the reported pool tuple resolves through the Thick factory and that the reported gauge is recognized by the voter. Both checks validate objects named by the target, but neither proves that `target` is the canonical NFPG associated with them. Any contract can therefore return values copied from a genuine NFPG and receive the Equalizer connector.

This classification has a custody-sensitive consequence. For a non-in-place compound, `NftFarmStrategy.compoundFor()` reaches `_compound()`, which calls `_withdrawNft()`, performs the zap, and then calls `_depositNft()`. Through the classified connector, `EqualizerNfpgConnector.withdrawNft()` calls `target.withdraw(tokenId)`, while `depositExistingNft()` transfers the NFT directly from the Sickle to `target`. A metadata-compatible contract can make `withdraw()` a no-op, implement `IERC721Receiver`, and receive the NFT during the deposit step. No postcondition confirms that the final target is canonical or that the expected staked ownership relationship was established.

The automated `Automation.compoundFor()` entrypoint nevertheless requires owner consent and an authorized keeper holding `NFT_COMPOUND`. The keeper is trusted to construct the execution parameters, including the farm address, so supplying such a target is not a privilege escalation under the stated threat model. The issue is the logical mismatch between the registry's intended classification and what it actually proves: connector discovery accepts structurally compatible impostors instead of authenticating Equalizer NFPG deployments.

Impact

Informational.

Recommendation

Authenticate the target address through a canonical Equalizer relationship instead of relying on metadata returned by the target. If the factory or voter exposes the canonical NFPG for a pool or gauge, derive that address independently and require exact equality with `target`. If no authoritative reverse lookup exists, explicitly register supported NFPG addresses in `ConnectorRegistry` rather than recognizing them heuristically.

As defense in depth, bind the owner-approved farm address and pool index to the NFT automation settings and require the keeper-supplied farm to match. After a withdrawal/deposit round trip, also verify the expected custody postcondition through a canonical connector—for example, that the Sickle again owns the unstaked NFT or that the authenticated NFPG reports the Sickle as the staked owner. These checks reduce the blast radius of routing mistakes without changing the trusted-keeper model.

Developer Response

Documented in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts#894), commit d4701c3b.

- Registry contents are admin-controlled. Recorded in `docs/AutomationKeeperTrust.md` as a trusted input.

2.9.4 Rebalance can strand NFT automation revocations on obsolete token IDs

`setNftConfig()` and `unsetNftConfig()` silently accept obsolete token IDs. If a rebalance migrates settings first, a pending owner update succeeds on the empty old key while the live replacement retains the previous policy.

Technical Details

When an NFT position is rebalanced, the system mints a new token ID and migrates configuration from the old key to the new one. The multi-compound registry stores per-position settings in `nftConfigMap` using a key derived from `(sickle, nftManager, tokenId)`. During a rebalance, `NftSettingsRegistry.transferNftSettings()` calls `MultiCompoundSettingsRegistry.onRebalance()`, which copies the configuration from the old key to the new key and deletes the old entry.

An owner can call `unsetNftConfig()` or `setNftConfig()` to revoke or update a position's automation policy. These functions operate on the key provided by the caller without checking whether that key still represents the live position. If a keeper-triggered rebalance executes before the owner's transaction, the configuration migrates to the new token ID while the old key becomes empty. When the owner's transaction then executes, `unsetNftConfig()` deletes the already-empty old slot and emits `NftConfigUnset(oldKey)`, while `setNftConfig()` writes the updated policy only to the obsolete key. The live position at the new key retains the original policy.

The same behavior affects the core NFT settings managed by `NftSettingsRegistry`, as its own `unset` operation similarly does not verify the key's current status. When automation selectors are registered for generic execution through `Automation.executeFor()`, a keeper can batch the rebalance and a subsequent multi-compound operation that consumes the migrated configuration in a single transaction, using the predictable new token ID. The validation in `MultiFarmStrategy._validateMultiCompoundFor()` and `_checkDepositTarget()` reads the configuration at the new key and accepts it, allowing the operation to proceed before the owner's revocation transaction can execute.

Impact

Informational.

Recommendation

Address owner policy by a stable logical position identifier rather than by the rebalance-unstable token ID. One approach is to assign each position a monotonically increasing settings ID and map the changing `NftKey` to that ID, with rebalances updating only the key-to-ID mapping while owners mutate the configuration by stable ID. Alternatively, store a revocation epoch per `(sickle, nftManager)` or per logical position, include that epoch in the configuration, and require both migration and automated execution to match the current epoch; owner revocation then increments the epoch, invalidating both the old and migrated entries regardless of transaction ordering.

As a minimum defense, require `setNftConfig()` and `unsetNftConfig()` to accept an expected current key or version and revert when the supplied key does not match or is already empty, so stale transactions fail visibly and can be retried against the replacement key. A robust solution must also check a stable revocation nonce during every automated action to prevent a keeper from consuming the migrated policy after the owner has incremented the epoch, even when rebalance and consumption are batched in a single transaction.

Developer Response

Partially fixed in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit d3727657.

- Revoking a stale or absent opt-in is now rejected rather than silently succeeding.
- The general case — a revocation targeting a token ID a rebalance has since replaced — is unchanged.

2.9.5 Automation unnecessarily exposes its internal batching helper

`Automation` inherits the external `multicall()` entrypoint even though its batching functionality is intended only for self-calls from the automation entrypoints. This unnecessarily expands the contract's external surface, couples automation to the Sickle caller whitelist, and adds avoidable execution overhead.

Technical Details

The batched variants of `compoundFor()`, `harvestFor()`, `exitFor()`, and `rebalanceFor()` build arrays of targets and calldata before invoking `this.multicall(targets, data)`. Because `multicall()` is inherited from `NonDelegateMulticall`, it remains externally callable and authorizes callers through `SickleRegistry.isWhitelistedCaller()`.

No external caller is expected to use `Automation.multicall()`, yet every caller allowed by the registry-wide Sickle security model can reach it. This creates an unnecessary permissioned entrypoint whose behavior and safety depend on the Sickle caller and target whitelists. It also requires the `Automation` contract itself to remain configured as a whitelisted caller so that its self-calls succeed. Registry changes can therefore unexpectedly break automation or broaden access to a batching surface that automation does not need to expose.

The external self-call also incurs ABI encoding, calldata copying, and an unnecessary caller-whitelist lookup compared with an internal batching helper.

Impact

Informational.

Recommendation

Move the batching loop into an internal helper and call it directly from the automation entrypoints. If preserving the external self-call is required, restrict `multicall()` in `Automation` to `msg.sender == address(this)` instead of relying on `isWhitelistedCaller()`.

Developer Response

Fixed in [vfat-io/sickle-contracts#895](#), landed on `main` via #891, commit b21233d3.

- `multicall()` is disabled externally with `MulticallDisabled`; batching moved to an internal `_multicall()`.

2.9.6 Wildcard permissions silently expand when future actions are introduced

`AutomationPermissions.ALL` is defined as `type(uint256).max`, so it includes permission bits that are not yet assigned. An automator or Sickle owner using this wildcard can therefore gain or grant access to future automation categories without a new review or explicit consent transaction.

Technical Details

`ALL` was changed from a bitmap of defined permissions to `type(uint256).max`.

`Automation.setApprovedAutomator()` assigns this value by default.

An automator granted `ALL` therefore automatically receives every permission bit introduced in the future, without a new administrative review or grant. Owners who use the same wildcard for `actionConsent` likewise pre-consent to future action categories whose risk and behavior did not exist when consent was given.

The separate owner-consent check reduces current exposure when owners consent only to explicit bits, but it does not protect wildcard consent.

Impact

Informational.

Recommendation

Define `ALL` as the union of currently supported permission bits, such as `(1 << 17) - 1`, and require explicit updates when new categories are introduced. Avoid exposing an unbounded wildcard as the default for automator permissions or owner action consent.

Developer Response

Acknowledged, no change. Wildcard permissions are ours, not users'. Introducing a new action bit is a deliberate deployment step, not something that happens implicitly.

2.9.7 Automation leaves economic execution choices to trusted keepers

Automation verifies the Sickle owner's action consent, automator authorization, and several action-specific position constraints. The opt-in is not a complete specification of every execution detail or economic outcome, so the approved automator or keeper remains a trusted role with significant responsibility over the positions that opt into automation.

Technical Details

The authorization layer checks whether a caller may perform an automation category for a Sickle, while downstream strategies validate the position and selected integration to varying degrees. Many arguments remain keeper-supplied because their safety or economic quality is difficult to fully express and enforce on-chain. Owners should therefore understand an automation opt-in as trust in the keeper to submit valid, conservative, and policy-compatible parameters.

Examples of keeper-controlled choices or outcomes include:

- `SwapParams` fields such as `tokenIn`, `amountIn`, `tokenApproval`, `router`, and `minAmountOut`, together with opaque aggregator `extraData` that may encode recipients and other routing choices.
- Rebalance target ticks and price context. Exact tick checks can confirm the submitted range matches a pre-operation spot-tick calculation, but they do not by themselves prove that the range remains economically appropriate after keeper-controlled swaps or market movement.
- Desired amounts, minimum amounts, swap routes, and other slippage or price-impact parameters. A call may satisfy structural pool and position checks while providing little value, obtaining poor execution, or leaving the position immediately eligible for another operation.
- Keeper-selected fee-token and sweep-token arrays, which can omit expected fee assets or include unrelated balances where the downstream path does not derive the lists from actual results.
- Opaque integration-specific claim, deposit, withdrawal, and callback data, plus generic `executeFor` registrations that may select a route without typed vault callbacks or their associated safeguards.

These examples are not additional claims that every path is exploitable. They illustrate the boundary between invariants that can be checked on-chain and execution quality, route safety, and integration behavior that still require trusted keeper operation, careful off-chain validation, and monitoring.

Impact

Informational.

Recommendation

Document the keeper trust assumptions for every automation category and require keepers to simulate the exact calldata, validate recipients and spend limits, use conservative minimums, and check the post-operation state before submitting transactions.

Developer Response

Documented in [vfat-io/sickle-contracts#894](https://github.com/vfat-io/sickle-contracts/pull/894), commit d4701c3b.

- `docs/AutomationKeeperTrust.md` states the enforced/trusted boundary and lists which config fields are advisory keeper inputs rather than on-chain constraints.

2.9.8 Some automated NFT harvest paths use the manual fee selector

Technical Details

The strategy defines distinct `Harvest` and `HarvestFor` fee selectors. The test configuration also assigns different rates to them: `90` and `120` respectively.

Although `NftFarmStrategy.harvestFor()` correctly calls `NftFarmStrategy._harvest()` with `HarvestFor`, the fee argument is discarded whenever swaps are present:

```
1 function _harvest(
2     Sickle sickle,
3     NftPosition calldata position,
4     NftHarvest calldata params,
5     bytes4 fee
6 ) private {
7     if (params.swaps.length > 0) {
8         _claimAndSwap(sickle, position, params);
9     } else {
10        _claim(sickle, position, params.harvest, fee);
11    }
12    // ...
13 }
```

`NftFarmStrategy._claimAndSwap()` hardcodes the manual selector:

```
1 data[2] = abi.encodeCall(
2     IFeesLib.chargeFees,
3     (address(this), NftFarmStrategyFees.Harvest, params.outputTokens)
4 );
```

As a result, automated `harvestFor()` and `rebalanceFor()` operations containing swaps pay the `Harvest` fee rather than `HarvestFor`.

In addition, `NftFarmStrategy._exit()` always invokes `NftFarmStrategy._harvest()` with `Harvest`, even when it is reached through `NftFarmStrategy.exitFor()`.

Impact

Informational.

Recommendation

Pass the selected fee through every helper:

```
1 if (params.swaps.length > 0) {
2     _claimAndSwap(sickle, position, params, fee);
3 } else {
4     _claim(sickle, position, params.harvest, fee);
5 }
```

Similarly, make `_exit()` accept a harvest-fee argument and pass `HarvestFor` from `exitFor()` while preserving `Harvest` for manual exits.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](#), commit 65970bf9.

- Automated harvest-with-swaps and `exitFor()` now charge `HarvestFor` rather than the manual selector.

2.9.9 Automated NFT deposits can add vault funds to an unrelated position manager

Technical Details

`NftFarmStrategy.depositFor()` validates the automation settings using the supplied `position.nft` and `position.tokenId`, but it only compares the add-liquidity `tokenId` with the validated position.

There is no equivalent comparison between `increaseParams.zap.addLiquidityParams.nft` and `position.nft`. `NftFarmVault.beforeDeposit()` has the same incomplete check: it verifies the `tokenId`, then approves the Sickle to pull the vault's `token0` and `token1`. Consequently, an authorized keeper can validate position (manager A, tokenId X) but execute the add-liquidity operation against (manager B, tokenId X).

`Automation.depositFor()` is permissioned: the caller must be a global or custom automator holding the `NFT_DUST_DEPOSIT` permission, and the Sickle must have consented to that action.

Impact

Informational.

Recommendation

Bind both components of the add-liquidity target to the validated position:

```
1 NftAddLiquidity calldata addLiquidity =
2     increaseParams.zap.addLiquidityParams;

4 if (
5     addLiquidity.nft != position.nft
6     || addLiquidity.tokenId != position.tokenId
7 ) {
8     revert DepositPositionMismatch();
9 }
```

Apply the same validation in `NftFarmVault.beforeDeposit()` as defense in depth.

Developer Response

Fixed in [vfat-io/sickle-contracts#894](#), commit 0f712546.

- `depositFor()` binds the zap's `addLiquidityParams.nft / tokenId` to the validated position.
- Added regression tests for the mismatch.



Sickle Automation

Completed 2026-08-05