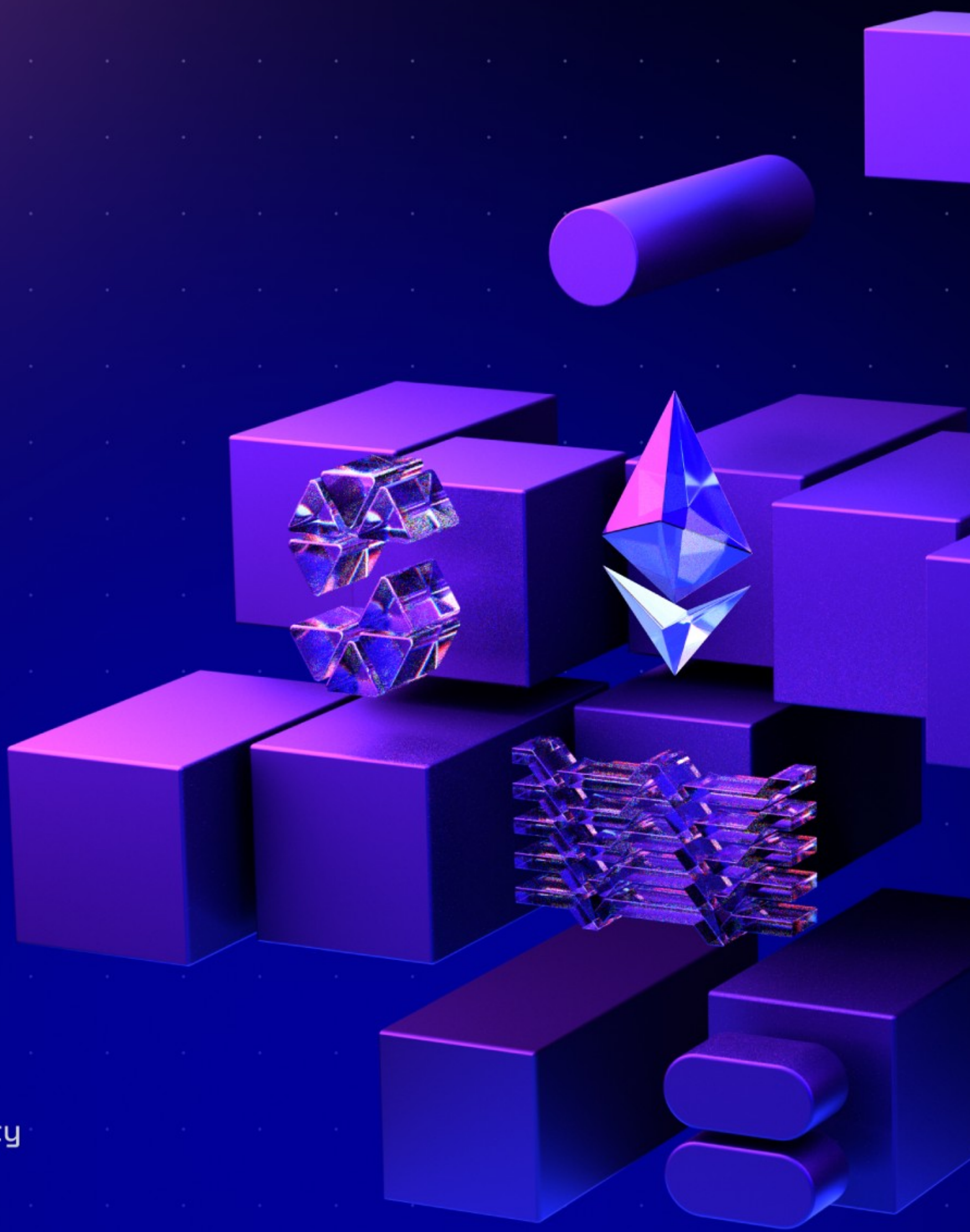
ackee
blockchain security

Vfat

Sickle

9.5.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	28.03.2025
1.0	Final Report	31.03.2025
1.1	Fix Review Report	09.05.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.

Rohanske nabrezi 717/4

186 00 Prague, Czech Republic

https://ackee.xyz

hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**

   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**

   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**

   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**

   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**

   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or `configuration`, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or `configuration` was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Andrey Babushkin	Lead Auditor
Štěpán Šonský	Lead Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Vfat is a yield aggregator, utilizing Sickle smart contract wallet for yield farming. It reduces complex operations such as entering/exiting positions, compounding, or rebalancing into single transactions.

Revision 1.0

Vfat engaged Ackee Blockchain Security to perform a security review of the Vfat protocol with a total time donation of 18 engineering days in a period between March 4 and March 28, 2025, with Andrey Babushkin as the lead auditor.

The audit was performed on the commit `357593f`[1] and the scope was the following:

- `contracts/Automation.sol`
- `contracts/ConnectorRegistry.sol`
- `contracts/NftSettingsRegistry.sol`
- `contracts/PositionSettingsRegistry.sol`
- `contracts/Sickle.sol`
- `contracts/SickleFactory.sol`
- `contracts/SickleRegistry.sol`
- `contracts/governance/SickleMultisig.sol`
- `contracts/libraries/FeesLib.sol`
- `contracts/libraries/NftSettingsLib.sol`
- `contracts/libraries/NftTransferLib.sol`
- `contracts/libraries/PositionSettingsLib.sol`

- `contracts/libraries/SwapLib.sol`
- `contracts/libraries/TransferLib.sol`

For completeness, it was also necessary to review the following parent contracts:

- `base/Admin.sol`
- `base/Multicall.sol`
- `base/NonDelegateMulticall.sol`
- `base/SickleStorage.sol`
- `base/TimelockAdmin.sol`

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- ensuring the arithmetic of the system is correct;
- detecting possible reentrancies in the code;
- checking the safety of the `delegatecall` usage;
- ensuring access controls are not too relaxed or too strict;
- checking the correctness of the upgradeability implementation; and
- looking for common issues such as data validation.

Our review resulted in 31 findings, ranging from Info to High severity. The most severe finding H1 allows an admin (malicious or compromised) to drain all user wallets. The medium severity issue M1 allows front-running of the `setReferralCode` function. Most findings are warnings that point to various missing validations, code quality issues, and best practices violations.

Ackee Blockchain Security recommends Vfat:

- mitigate the admin trust issues with whitelisted callers;
- resolve the front-running risk;
- avoid EOA reverts in the `ConnectorRegistry` contracts;
- add missing data validation;
- address all other reported issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

The review was done on the given commit `1c20e7e`[2]. The scope of the fix review was limited to issues found in the previous revision and no other code changes were audited. 20 issues were fixed, 3 issues fixed partially, 7 issues acknowledged, and 1 issue (H1) was invalidated by the client.

[1] full commit hash: `357593f493ef063706365639a047fab70cd7431c`

[2] full commit hash: `1c20e7e195c3c4fe621474183d67aa3b21bb54fa`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- `Description`
- `Exploit scenario` (if severity is low or higher)
- `Recommendation`
- `Fix` (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	1	1	16	12	31

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Whitelisted callers can perform delegatecall on every Sickle	High	1.0	Reported
M1: Referral code setter can be front-run	Medium	1.0	Acknowledged
L1: Non-contract registries can cause reverts	Low	1.0	Fixed
W1: Incomplete data validation for NFT positions	Warning	1.0	Partially fixed
W2: Duplicate Sickle retrieval	Warning	1.0	Acknowledged

Finding title	Severity	Reported	Status
W3: Potential underflow or overflow in tick range calculation	Warning	1.0	Fixed
W4: Variable shadowing	Warning	1.0	Fixed
W5: Insufficient data validation in the `PositionSettingsRegistry` contract	Warning	1.0	Fixed
W6: Incorrect price calculation in PositionSettingsRegistry	Warning	1.0	Fixed
W7: Incorrect usage of Initializable	Warning	1.0	Fixed
W8: Variable naming convention	Warning	1.0	Fixed
W9: One-step ownership transfer	Warning	1.0	Acknowledged
W10: Duplicate tokens in `feeTokens` array can lead to inconsistent fee calculation	Warning	1.0	Acknowledged
W11: Inconsistent handling of ETH and WETH across the `FeesLib` contract	Warning	1.0	Acknowledged
W12: Ambiguous handling of the native value in the `SwapLib` contract	Warning	1.0	Acknowledged
W13: Misleading inheritance	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W14: No input array length validation	Warning	1.0	Fixed
W15: No data validation on registry adding and updates	Warning	1.0	Partially fixed
W16: Missing zero address validation	Warning	1.0	Acknowledged
I1: Duplicate code	Info	1.0	Fixed
I2: Usage of magic constants	Info	1.0	Fixed
I3: Unconsolidated storage variable definitions	Info	1.0	Fixed
I4: Redundant storage variable	Info	1.0	Fixed
I5: Mapping `isCustomRegistry` is redundant	Info	1.0	Fixed
I6: Inconsistent function naming convention	Info	1.0	Fixed
I7: Typographical error in function comment	Info	1.0	Fixed
I8: Misleading error name	Info	1.0	Fixed
I9: Unused errors	Info	1.0	Fixed
I10: Redundant function	Info	1.0	Fixed
I11: Missing duplicate registry validation	Info	1.0	Partially fixed
I12: Errors in documentation	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member’s Name	Position
Andrey Babushkin	Lead Auditor
Martin Veselý	Auditor
Štěpán Šonský	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The protocol is a farming automation system where users receive unique `Sickle` instances deployed deterministically through the `SickleFactory` using CREATE2. Approved Automators can execute compound, harvest, and rebalance operations on users’ behalf. Position settings are managed through the `PositionSettingsRegistry` for standard positions and the `NftSettingsRegistry` for NFT positions. The protocol features a fee system capped at 5% and specialized operation libraries. Protocol governance is controlled through a `SickleMultisig` contract with configurable thresholds and signer management. Integration with DeFi protocols is handled through an updatable Connector registry system.

Trust Model

The protocol requires users to trust administrators who control critical parameters (fees, whitelists, Connector updates) and Automators who execute operations on their behalf. While users control their Sickle instances and position settings, the system maintains centralized control points. Trust risks are partially mitigated through hardcoded limits and multisig requirements; however, users must accept risks of centralized control and potential transaction manipulation by Automators who can control transaction timing.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Whitelisted callers can perform delegatecall on every Sickle

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Sickle.sol	Type:	Trust model

Description

The `Sickle` contract inherits from `Multicall`, which implements the multicall logic using `delegatecall`. The `Multicall` contract verifies if the caller is whitelisted by calling the `isWhitelistedCaller` function on the `SickleRegistry` contract. Similarly, the call target is verified by calling the `isWhitelistedTarget` function on the `SickleRegistry` contract. Since the registry is a singleton contract and it is stored in the immutable variable in the `Multicall` contract, the whitelisted callers can perform operations on every deployed `Sickle` contract. This may be misused either by a malicious caller that was mistakingly whitelisted by the `SickleRegistry` admin, or by the admin themselves by adding an arbitrary malicious target and a malicious caller. For example, this may hapeen in the case of private key leakage of the admin account. In this case, all user wallets can be drained resulting in loss of all users' assets.

Exploit scenario

1. Alice, the protocol admin, leaks the private keys for the admin address of the `SickleRegistry` contract.
2. Bob, who receives the unauthorized access, adds himself to the callers whitelist by calling `setWhitelistedCallers([0xBob], true)` on the `SickleRegistry` contract.
3. Similarly, Bob calls `setWhitelistedTargets([0xDrainer], true)` to whitelist a

drainer contract in the registry.

4. Finally, Bob calls `multicall()` on all users' deployed Sickle contracts to perform `delegatecall` to the drainer contract and steal all assets from all users' wallets.

This whole operation can be done in one transaction, and cannot be mitigated by the on-chain analysis and quick response.

Recommendation

Implement one or more of the following mitigations:

1. Implement a time-delay mechanism for adding new whitelisted callers and targets. This gives users time to react if suspicious addresses are whitelisted.
2. Split the admin role into multiple roles with different permissions:
   - One role for managing whitelisted callers;
   - One role for managing whitelisted targets;
   - Require multiple signatures to make changes.
3. Add an emergency pause functionality that can quickly disable all delegatecalls across all Sickle contracts.
4. Remove the `delegatecall` entirely and implement specific approved functions directly in the Sickle contract.
5. Move whitelisting control to individual Sickle contract owners:
   - Each Sickle owner maintains their own whitelist of approved callers and targets.
   - Registry only stores global defaults that owners can accept or reject.
   - Owners can opt-in to global whitelist or manage their own permissions.

Update 1.1

The issue was invalidated by the client with the following comment.

> *The Optimism deployment as audited is already using the standard practice of a timelocked multisig, therefore this is not possible as described.*
>
> — Vfat

[Go back to Findings Summary]

M1: Referral code setter can be front-run

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	SickleRegistry.sol	Type:	Front-running

Description

The `SickleRegistry` contract allows for assigning an address to each referral code. If an address is already assigned, it cannot be changed:

Listing 1. Excerpt from SickleRegistry

```
function setReferralCode(bytes32 referralCode) external {
    if (referralCodes[referralCode] != address(0)) {
        revert InvalidReferralCode();
    }

    referralCodes[referralCode] = msg.sender;
    emit SickleRegistryEvents.ReferralCodeCreated(referralCode, msg.sender);
}
```

The `setReferralCode` function, however, can be front-run and a malicious referer can be assigned. This way, the malicious address can steal all the referral rewards.

Exploit scenario

1. Alice creates a referral code by generating a random `bytes32` sequence with the intent of sharing the code with others and earn rewards for each user who uses this code.
2. She calls `setReferralCode` to assign her address to the generated code.
3. Bob observes this transaction in the mempool and creates an identical transaction with the same parameters and higher gas value.

4. The transaction gets included in the block before the initial transaction by Alice.
5. Bob's address is now assigned to the referral code.
6. If Alice doesn't notice the reverted transaction and shares the code, Bob will receive all the rewards instead of Alice.

Recommendation

Implement one of these approaches:

- Use a commit-reveal pattern where users first commit to a hash of their referral code and address, then reveal it after a time delay.
- Make referral codes unique to each user by incorporating the sender's address into the referral code generation.
- Add a small registration fee to make front-running economically unattractive.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *Referral codes are not currently being used and the main chains used do not have a front-running issue, will be looked into if we decide to activate referrals.*
>
> — Vfat

Go back to Findings Summary

L1: Non-contract registries can cause reverts

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	ConnectorRegistry.sol	Type:	Data validation

Description

In the `ConnectorRegistry` contract, the `connectorOf` function returns the address of the connector for the provided `target`. First, the contracts checks if the connector is registrered in the `connectors_` mapping. If it is, the connector address is returned to the callee. Otherwise, the algorithms iterates over all items in the `customRegistries` array and calls the `connectorOf(target)` function on each registry. The `hasConnector` function has similar logic.

The call is performed in the try-catch clause. If the external call reverts, the error is caught and ignored. However, there are cases that are not caught by the try-catch statement, and one of these cases is when the callee is an EOA.

If by any mistake an address without code is added to a list of custom registries during the `addCustomRegistry` call, the `connectorOf` and `hasConnector` functions will revert.

Exploit scenario

1. Alice, the contract admin, calls `addCustomRegistry` with the `registry` parameter set to the `0x1111111111111111111111111111111111111111` address. The address is not a contract and does not have any code. The address is successfully added to the system as a registered custom registry.
2. Bob calls the `connectorOf` function for any target.
3. The `connectors_` mapping does not have a connector address set for the

provided target.

4. The function begins to iterate over the `customRegistries` array and calls the `connectorOf(target)` function on the stored `0x1111111111111111111111111111111111111111`.
5. The call reverts without being caugth by the try-catch clause.
6. The reason of the revert is not clear to Bob.

Recommendation

Add validation of the provided addresses to the `addCustomRegistry` function. Make sure address is a contract. Implement off-chain verification and monitoring to avoid situations like this.

Fix 1.1

The issue was fixed.

[Go back to Findings Summary]

W1: Incomplete data validation for NFT positions

Impact:	Warning	Likelihood:	N/A
Target:	NftSettingsRegistry.sol	Type:	Data validation

Description

The `NftSettingsRegistry` lacks comprehensive data validation for position settings for multiple scenarios:

- No validation that `triggerTickLow` >= `MIN_TICK` and `triggerTickHigh` <= `MAX_TICK`;
- No validation that `triggerTickLow` < `triggerTickHigh`;
- No validation that `exitTokenOutLow` and `exitTokenOutHigh` are valid contract addresses (not EOAs or zero addresses) when `autoExit` is **enabled**;
- No validation that `harvestTokenOut` is a valid contract address (not EOAs or zero address) when `rewardBehavior` is `Harvest`;
- No minimum/maximum boundaries for `bufferTicksBelow` and `bufferTicksAbove`;
- No validation for `delayMin` being within reasonable bounds;
- No validation that the specified pool address is valid and corresponds to the NFT position's pool;
- No checks to ensure exit triggers are outside of rebalance cutoffs for logical operation;
- No validation that token addresses are actual contracts;
- Incomplete validation of token address relationships with pool tokens;
- No validation that configured ticks align with the pool's tick spacing requirements;

- Only width is validated but not the actual tick values;
- No validation that the NFT position exists and belongs to the caller;
- If both `autoExit` and `autoRebalance` are `true`:
- `triggerTickLow` should be ⇐ `cutoffTickLow` (exit before stop-loss), and
- `triggerTickHigh` should be >= `cutoffTickHigh` (exit before stop-loss);
- The conflict when `rewardBehavior` == `Compound` and `autoExit` == `true` is not handled.

Recommendation

Evaluate all possible scenarios and create a comprehensive list of invariants. Implement the missing validations.

Partial solution 1.1

The issue was partially fixed with the following comment.

> - *Token/EOA address validation is deemed unnecessary*
> - *DelayMin has reasonable bounds by virtue of being a uint8*
> - *Pool address validation / token address relationship with pool tokens is not trivial so deemed impractical*
> - *Configured ticks do not need to align with tick spacing*
> - *Exit before rebalance cutoff is not a conflict*
> - *Compound and auto-exit do not conflict*
>
> *In general, mistakes in NFT settings are not critical since they only require an update, we try to strike a balance between warning users early (by reverting on obvious errors) and not requiring too much computation.*

> — Vfat

Go back to Findings Summary

W2: Duplicate Sickle retrieval

Impact:	Warning	Likelihood:	N/A
Target:	NftSettingsRegistry.sol	Type:	Code quality

Description

The `NftSettingsRegistry` and `PositionSettingsRegistry` have the `_get_sickle_by_owner` function with the duplicated logic.

Recommendation

Move the `Sickle` retrieval to the `SickleFactory` contract.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *Duplication in two contracts is acceptable vs the cost of amending (redeploying all Sickles).*
>
> — Vfat

[Go back to Findings Summary]

W3: Potential underflow or overflow in tick range calculation

Impact:	Warning	Likelihood:	N/A
Target:	NftSettingsRegistry.sol	Type:	Overflow/Underflow

Description

In `NftSettingsRegistry`, the `validateRebalanceFor` function calculates the allowed tick range by calculating the bounds using the buffer:

Listing 2. Excerpt from NftSettingsRegistry

```
if (
    tick >= tickLower - int24(config.bufferTicksBelow)
        && tick < tickUpper + int24(config.bufferTicksAbove)
) {
    revert TickWithinRange();
}
```

Since there is no clipping for extreme values, it is possible for `tickLower - int24(config.bufferTicksBelow)` to underflow if `tickLower` is very small, or `tickUpper + int24(config.bufferTicksAbove)` to overflow if `tichUpper` is large.

Recommendation

Add bounds checking to ensure the buffer calculations cannot overflow/underflow.

Fix 1.1

The issue was fixed with the following comment.

> *Ticks and bufferTicksAbove/Below are within MIN_TICK / MAX_TICK now so underflow and overflow are no longer*

> *possible.*
>
> — Vfat

Go back to Findings Summary

W4: Variable shadowing

Impact:	Warning	Likelihood:	N/A
Target:	PositionSettingsRegistry.sol	Type:	Code quality

Description

In the `PositionSettingsRegistry` contract, the `constructor’s `timelockAdmin` argument shadows the storage variable inherited from the `TimelockAdmin` contract. This may lead to confusion during code review and maintenance, and could potentially cause bugs if the shadowed variable is accessed incorrectly.

Also:

- `SickleFactory::_deploy`, `admin` variable`;
- `SickleFactory::_getSickle`, `admin` variable`;
- `SickleFactory::predict`, `admin` variable`;
- `SickleFactory::sickles`, `admin` variable`;
- `SickleFactory::admins`, `admin` variable`;
- `SickleFactory::getOrDeploy`, `admin` variable`.

Recommendation

Rename the arguments to avoid shadowing storage variables.

Fix 1.1

The issue was fixed by renaming `admin` parameter to `owner`.

[Go back to Findings Summary]

W5: Insufficient data validation in the `PositionSettingsRegistry` contract

Impact:	Warning	Likelihood:	N/A
Target:	PositionSettingsRegistry.sol	Type:	Data validation

Description

The `PositionSettingsRegistry` contract lacks several critical data validations:

- the `constructor` does not validate zero addresses for input parameters;
- the `settings.pair` and `settings.router` addresses are not validated for zero addresses, and their relationship is not verified;
- the `farm.stakingContract` address is not validated for zero addresses; and
- the `harvestTokenOut`, `exitTokenOutLow`, and `exitTokenOutHigh` tokens may differ from pool tokens without a required swapping path.

Recommendation

Implement comprehensive data validation by:

- adding zero-address checks for all address parameters;
- validating relationships between interdependent parameters; and
- ensuring swapping paths exist when tokens differ from pool tokens.

Fix 1.1

The issue was fixed by adding extra checks.

[Go back to Findings Summary]

W6: Incorrect price calculation in PositionSettingsRegistry

Impact:	Warning	Likelihood:	N/A
Target:	PositionSettingsRegistry.sol	Type:	Arithmetics

Description

The `PositionSettingsRegistry` contract retrieves the current pool price by calling `getAmountOut` with the `amountIn` equal to 1 wei:

Listing 3. Excerpt from PositionSettingsRegistry

```
uint256 amountOut0 = connector.getAmountOut(
    GetAmountOutParams({
        router: address(settings.router),
        lpToken: address(settings.pair),
        tokenIn: token0,
        tokenOut: token1,
        amountIn: 1
    })
);

if (amountOut0 > 0) {
    price = amountOut0 * 1e18;
} else {
    uint256 amountOut1 = connector.getAmountOut(
        GetAmountOutParams({
            router: address(settings.router),
            lpToken: address(settings.pair),
            tokenIn: token1,
            tokenOut: token0,
            amountIn: 1
        })
    );
    if (amountOut1 == 0) {
        revert InvalidPrice();
    }
    price = 1e18 / amountOut1;
```

However, due to the small input amount, the `getAmountOut` may return zero for swapping in both directions when the pool price is close to 1. This is the case for stablecoin pairs. For instance, the USDC/USDT Uniswap v2 pool very similar reserves for both tokens, and calling the `getAmountOut` returns zero for both directions. In this case, the price validation on line 297 will not succeed, and the transaction will revert with the `InvalidPrice` error.

For example, the USDC/USDT pool on calling `getReserves()` returns `2359110923152` and `2363067862194`. If we call the `getAmountOut` function on the Uniswap v2 router with `amountIn = 1` and these reserves, the return value is zero for both swap directions.

Another problem is extreme values. If the return amount is larger than 1e18, the price calculation on line 300 will be zero, and the transaction will revert with the `InvalidPrice` error.

Recommendation

Since the `amountIn` is only used for price calculations and not the actual trade, use a larger value of `amountIn` to avoid the precision loss. Modify the price calculation accordingly. An alternative approach for pools with two tokens is to calculate the price based on the pool reserves directly.

Fix 1.1

The issue was fixed with the following comment.

> *Has since been refactored to a getPoolPrice call.*
>
> — Vfat

Go back to Findings Summary

W7: Incorrect usage of Initializable

Impact:	Warning	Likelihood:	N/A
Target:	Sickle.sol	Type:	Reinitialization

Description

The `Sickle` contract inherits from the `SickleStorage` and `Multicall` contracts. The `Multicall` constructor is marked with the `initializer` modifier, and the `Sickle` constructor also has the same keyword. Moreover, the `Sickle` contract also has a separate `initialize` function that is also marked `initializer`. When a new Sickle is deployed, the `Initializable` empty constructor is called first, then the `initializer` modifier on the `Multicall` constructor is invoked. Finally, the `initializer` modifier on the `Sickle` constructor is run, and the constructor is called.

The `Initializable` contract warns against this explicitly:

> *When used with inheritance, manual care must be taken to not invoke a parent initializer twice or to ensure that all initializers are idempotent. This is not verified automatically as constructors are by Solidity.*

The implemented approach will lead to multiple issues:

- The initialization logic will be executed multiple times;
- The initialization functions will be callable when they shouldn't be;
- Potential unexpected behavior in the proxy upgrade pattern.

The correct approach is to have one initializer per contract. If base contracts are abstract or expected to be inherited from, the `onlyInitializing` modifier should be used instead of `initializer`. The constructors are not initializers, and to prevent the unintended usage of the initializer, the `_disableInitializers` function must be called. To implement this approach, use the following steps:

1. Remove the `initializer` keyword from the `Multicall` constructor.
2. If any additional initializing logic must be added in the `Multicall` contract, create a separate `_Multicall_initialize` function on the `Multicall` contract and mark it with the `onlyInitializing` keyword. Make sure you call this function in all child contracts.
3. Remove the `initializer` keyword from the `Sickle` constructor.
4. Add a call to the `_disableInitializers` function in the `Sickle` constructor.
5. If the `Sickle` contract is inherited by other contracts, make sure the `_Sickle_initialize` function is `internal` and mark it with `onlyInitializing` keyword.

Recommendation

Implement the correct approach to the `Initializable` contracts.

Fix 1.1

The issue was fixed by the client with the following comment.

> *Multicall no longer has an initializer.*
>
> — Vfat

[Go back to Findings Summary]

W8: Variable naming convention

Impact:	Warning	Likelihood:	N/A
Target:	SickleFactory.sol	Type:	Code quality

Description

The `SickleFactory` contract has the `_referralCodes` that is marked `public` but its name starts with the underscore, which does not follow the naming convention.

Recommendation

Based on the project requirements, either rename the variable to `referralCodes` or make it `private`/`internal`.

Fix 1.1

The issue was fixed.

[Go back to Findings Summary]

W9: One-step ownership transfer

Impact:	Warning	Likelihood:	N/A
Target:	SickleFactory.sol	Type:	Access control

Description

The `SickleFactory` contract inherits from `Admin` where the `admin` role is stored. The `setAdmin` function of the `Admin` contract sets a new admin address in one step. If the address is set incorrectly, there is a risk of losing access to the contract.

Recommendation

Implement two-step admin priviledge transfer by using the `Ownable2Step` contract from the OpenZeppelin framework, or by implementing an in-house solution.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *This is not a single step as we use a multisig as admin.*
>
> — Vfat

Go back to Findings Summary

W10: Duplicate tokens in feeTokens array can lead to inconsistent fee calculation

Impact:	Warning	Likelihood:	N/A
Target:	FeesLib.sol	Type:	Data validation

Description

The `chargeFees` function of the `FeesLib` contract calls the `chargeFee` function in a loop with the `feeBasis` parameter set to zero:

Listing 4. Excerpt from FeesLib

```
for (uint256 i = 0; i < feeTokens.length;) {
    chargeFee(strategy, feeDescriptor, feeTokens[i], 0);
    unchecked {
        i++;
    }
}
```

The Sickle contract calls functions of the `FeesLib` contract using `delegatecall`. Therefore, the `chargeFee` function then uses the balance of the caller for the fee token as the `feeBasis`:

Listing 5. Excerpt from FeesLib

```
if (feeBasis == 0) {
    if (feeToken == ETH) {
        uint256 wethBalance = weth.balanceOf(address(this));
        if (wethBalance > 0) {
            weth.withdraw(wethBalance);
        }
        feeBasis = address(this).balance;
    } else {
        feeBasis = IERC20(feeToken).balanceOf(address(this));
    }
}
```

If the input `feeTokens` array, the input parameter to the `chargeFees` function, has duplicates, the charged fee for later calls to the `chargeFee` function will be influenced by the former calls, making the fee structure of the protocol dependent on the order in which `feeTokens` are provided. This reduces the transparency of the fee scheme of the protocol.

Recommendation

Make sure the input `feeTokens` array does not have duplicates.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *Acknowledged but is checked offchain.*
>
> — Vfat

[Go back to Findings Summary]

W11: Inconsistent handling of ETH and WETH across the `FeesLib` contract

Impact:	Warning	Likelihood:	N/A
Target:	FeesLib.sol	Type:	Logic error

Description

The `FeesLib` contract inconsistently handles native ETH and wrapped ETH (WETH) tokens across different functions, creating potential confusion and calculation errors:

1. In the `chargeFee` function, when handling native ETH, the code unwraps any WETH balance and combines it with the native balance for fee calculation:

Listing 6. Excerpt from FeesLib

```
if (feeToken == ETH) {
    uint256 wethBalance = weth.balanceOf(address(this));
    if (wethBalance > 0) {
        weth.withdraw(wethBalance);
    }
    feeBasis = address(this).balance;
```

2. However, when WETH address is provided directly to `chargeFee`, it's treated as a standard ERC-20 token, ignoring any native ETH balance:

Listing 7. Excerpt from FeesLib

```
    feeBasis = address(this).balance;
} else {
    feeBasis = IERC20(feeToken).balanceOf(address(this));
}
```

3. The `getBalance` function takes the opposite approach - when checking ETH balance, it returns only the WETH balance and ignores native ETH:

Listing 8. Excerpt from FeesLib

```
function getBalance(
    Sickle sickle,
    address token
) public view returns (uint256) {
    if (token == ETH) {
        return weth.balanceOf(address(sickle));
    }
    return IERC20(token).balanceOf(address(sickle));
}
```

This inconsistent treatment of ETH/WETH may lead to incorrect fee calculations, unexpected behavior, and implementation errors in contracts using this library.

Recommendation

Standardize the handling of ETH and WETH across the contract. Decide on a consistent approach and ensure all functions follow the same logic for ETH/WETH handling.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *These functions serve different purposes. chargeFees works both inbound and outbound, in the inbound case it will be charging directly on ETH (user sends ETH → gets charged fees). In the outbound case it has WETH which it needs to withdraw to ETH first before charging fees (and then sending the remainder back to the user).*
>
> — Vfat

Go back to Findings Summary

W12: Ambiguous handling of the native value in the `SwapLib` contract

Impact:	Warning	Likelihood:	N/A
Target:	SwapLib.sol	Type:	Logic error

Description

The `SwapLib` contract implements two swapping functions: swap and `swapMultiple`. The `swap` function is marked `payable`, while `swapMultiple` is not. Moreover, the internal `_swap` function does not handle the native token as the input token. If the zero address or the `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` value (which is used in the FeesLib contract) is passed in the `swapParams.tokenIn` input parameter, the call will revert either on getting the balance:

Listing 9. Excerpt from SwapLib

```
if (swapParams.amountIn == 0) {
    swapParams.amountIn = IERC20(tokenIn).balanceOf(address(this));
}
```

or on the approval:

Listing 10. Excerpt from SwapLib

```
SafeTransferLib.safeApprove(tokenIn, swapParams.router, 0);
```

Recommendation

Decide on using the native token as the input token. If the support is intended, wrap the native value into the WETH token in the `_swap` function and implement tracking of `msg.value` for all swaps in the `swapMultiple` function. If the support is not required, check if the input token is a valid ERC-20 token.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *As audited the SwapLib contract only handles WETH swaps (0xEee is wrapped at an earlier step by TransferLib, and 0x000 is rejected).*
>
> — Vfat

Go back to Findings Summary

W13: Misleading inheritance

Impact:	Warning	Likelihood:	N/A
Target:	Automation.sol	Type:	Code quality

Description

The `Automation` contract inherits from `Admin` and `NonDelegateMulticall`. The `Admin` contract allows for setting an admin of the contract, which is set during the creation and used in `setApprovedAutomator` and `revokeApprovedAutomator` functions through the `onlyAdmin` modifier.

However, the second parent contract, `NonDelegateMulticall`, also inherits from `SickleStorage`, which in turn inherits from `Initializable`. This inheritance chain has multiple problems.

First, the `Initializable` contract is intended to be used for upgradeable contracts; however, `Automation` is not upgradeable. Having an initializer in a static contract is misleading and may become a source of mistakes in the future.

Second, the `SickleStorage` implements two privileged roles, `owner` and `approved`. These roles must be set up in the `_SickleStorage_initialize` function, which should be called in the initializer. The `NonDelegateMulticall` contract has the constructor marked with the `initializer` keyword, however, it does not call `_SickleStorage_initialize`. Therefore, the storage variables `owner` and `approved` remain uninitialized, taking up two storage slots. Moreover, since the default values of these variables are zero addresses, the `setApproved` function of the `SickleStorage` contract, which has the `onlyOwner` modifier, cannot be called by anyone, effectively becoming dead code.

Finally, the `Automation` contract implements four privileged roles: `admin`, `owner`, `approvedAutomator`, and `approved`. Two of these roles cannot be used and may lead to confusion of the users and developers who want to extend the functionality of the `Automation` contract in the future.

Recommendation

Analyze the inheritance chain. Clearly distinguish between upgradeable and non-upgradeable contracts. Modularize the protocol and follow the Separation of Concerns pattern. For example, the `NonDelegateMulticall` contract can be separated from the `SickleStorage` contract and be added to the inheritance chain of child contracts only if needed.

Fix 1.1

The issue was fixed with the following comment.

> *NonDelegeateMulticall no longer inherits from SickleStorage.*
>
> — Vfat

Go back to Findings Summary

W14: No input array length validation

Impact:	Warning	Likelihood:	N/A
Target:	ConnectorRegistry.sol, Automation.sol	Type:	Data validation

Description

1. The `setConnectors` and `updateConnectors` functions take two arrays, `targets` and `connectors`. These arrays must have the same length. However, there are no validations, which may lead to an `OutOfBounds` panic.
2. The `Automation` contract provides an interface for multicall to strategies. Each function takes multiple arrays as input parameters for these multicalls. These arrays must be of the same length. While most parameters are correctly validated, the following parameters lack such validation:
    - the `farms` parameter in the `harvestFor` function:

Listing 11. Excerpt from Automation

```
function harvestFor(
    IAutomation[] memory strategies,
    Sickle[] memory sickles,
    Farm[] memory farms,
    HarvestParams[] memory params,
    address[][] memory sweepTokens
) external onlyApprovedAutomator {
    uint256 strategiesLength = strategies.length;
    if (
        strategiesLength != sickles.length
            || strategiesLength != params.length
            || strategiesLength != sweepTokens.length
    ) {
        revert InvalidInputLength();
    }
```

- the `inPlace` parameter in the `compoundFor` function:

Listing 12. Excerpt from Automation

```
function compoundFor(
    INftAutomation[] memory strategies,
    Sickle[] memory sickles,
    NftPosition[] memory positions,
    NftCompound[] memory params,
    bool[] memory inPlace,
    address[][] memory sweepTokens
) external onlyApprovedAutomator {
    uint256 strategiesLength = strategies.length;
    if (
        strategiesLength != sickles.length
            || strategiesLength != positions.length
            || strategiesLength != params.length
            || strategiesLength != sweepTokens.length
    ) {
        revert InvalidInputLength();
    }
```

Recommendation

Add validation to ensure that the input arrays have the same length.

Fix 1.1

The issue was fixed by adding the length validation.

[Go back to Findings Summary]

W15: No data validation on registry adding and updates

Impact:	Warning	Likelihood:	N/A
Target:	ConnectorRegistry.sol	Type:	Data validation

Description

The `addCustomRegistry` function adds new custom registries to the `customRegistries` array and updates the `isCustomRegistry` mapping to mark the registry as added. However, the function lacks input validation and does not verify if the registry is already added. If the same registry is added multiple times, the array will contain duplicates, resulting in multiple indices allocated for the same custom registry in the `customRegistries` array.

An issue arises when the `updateCustomRegistry` function is called with any of the allocated indices. The function sets `isCustomRegistry` to `false` and replaces the element in `customRegistries` with the new address. However, any duplicates remain unaffected, creating a situation where a registry address exists in the `customRegistries` array while `isCustomRegistry` returns `false` for this address.

The `updateCustomRegistry` function does not validate if the input `index` is within the current bounds of the `customRegistries` array. If an invalid index is provided, the execution reverts with an `OutOfBounds` panic.

Furthermore, both the newly added registry address and the updated registry address can be zero addresses. While the `updateCustomRegistry` logic suggests that zero addresses are expected, the `isCustomRegistry` function returns `true` for zero addresses, which is misleading and incorrect.

Recommendation

Add the following validations: - Check if `isCustomRegistry` is already set for the registry address in the `addCustomRegistry` function. - Verify that the input `index` is less than `customRegistries.length` in the `updateCustomRegistry` function. - Implement zero address validation.

Partial solution 1.1

The issue was partially fixed with the following comment.

> *Added duplicate check in 2477a23dd7d25af191b67e47beff085e78affba8.Further validation checks are done offchain in the deployment script as these are admin functions.*
>
> — Vfat

Go back to Findings Summary

W16: Missing zero address validation

Impact:	Warning	Likelihood:	N/A
Target:	*/.sol	Type:	Data validation

Description

Multiple contracts in the codebase lack zero address validation for critical address parameters in constructors and setter functions. The absence of zero address validation could result in permanently broken contract functionality if zero addresses are inadvertently set. The following instances lack zero address validation:

In the `SickleFactory` contract:

Listing 13. Excerpt from SickleFactory

```
constructor(
    address admin_,
    address sickleRegistry_,
    address sickleImplementation_,
    address previousFactory_
) Admin(admin_) {
    registry = SickleRegistry(sickleRegistry_);
    implementation = sickleImplementation_;
    previousFactory = SickleFactory(previousFactory_);
}
```

- `admin_` parameter
- `sickleRegistry` parameter
- `sickleImplementation` parameter

In the `ConnectorRegistry` contract:

Listing 14. Excerpt from ConnectorRegistry

```
constructor(
```

```
    address admin_,
    address timelockAdmin_
) Admin(admin_) TimelockAdmin(timelockAdmin_) { }
```

- admin_ parameter
- timelockAdmin parameter

In the SickleMultisig contract:

Listing 15. Excerpt from SickleMultisig

```
constructor(address initialSigner) {
    // Initialize with only a single signer and a threshold of 1. The signer
    // can add more signers and update the threshold using a proposal.
    _addSigner(initialSigner);
    _setThreshold(1);
}
```

Listing 16. Excerpt from SickleMultisig

```
function _addSigner(address signer) internal changesSettings {
    if (isSigner(signer)) revert SignerAlreadyAdded();

    _signers.add(signer);

    emit SignerAdded(signer);
}
```

- initialSigner parameter
- signer parameter

In the NftSettingsRegistry contract:

Listing 17. Excerpt from NftSettingsRegistry

```
constructor(SickleFactory _factory, ConnectorRegistry _connectorRegistry) {
    factory = _factory;
    connectorRegistry = _connectorRegistry;
```

```
}
```

- `_factory` parameter
- `_connectorRegistry` parameter

In the `Automation` contract:

Listing 18. Excerpt from Automation

```
constructor(
    SickleRegistry registry_,
    address payable approvedAutomator_,
    address admin_
) Admin(admin_) NonDelegateMulticall(registry_) {
    _setApprovedAutomator(approvedAutomator_);
}
```

- `registry_` parameter
- `approvedAutomator_` parameter
- `admin_` parameter

In the `PositionSettingsRegistry` contract:

Listing 19. Excerpt from PositionSettingsRegistry

```
constructor(
    SickleFactory _factory,
    ConnectorRegistry connectorRegistry,
    address timelockAdmin
) TimelockAdmin(timelockAdmin) {
    factory = _factory;
    _connectorRegistry = connectorRegistry;
    emit ConnectionRegistrySet(address(connectorRegistry));
}
```

- `_factory` parameter
- `connectorRegistry` parameter

- `timelockAdmin` parameter

In the `FeesLib` library:

Listing 20. Excerpt from FeesLib

```
constructor(SickleRegistry registry_, WETH weth_) {
    registry = registry_;
    weth = weth_;
}
```

- `registry_` parameter
- `weth_` parameter

In the `SwapLib` library:

Listing 21. Excerpt from SwapLib

```
constructor(
    ConnectorRegistry connectorRegistry_
) {
    connectorRegistry = connectorRegistry_;
}
```

- `connectorRegistry_` parameter

In the `TransferLib` library:

Listing 22. Excerpt from TransferLib

```
constructor(IFeesLib feesLib_, WETH weth_) {
    feesLib = feesLib_;
    weth = weth_;
}
```

- `feesLib_` parameter
- `weth_` parameter

Recommendation

Add zero address validation checks for all critical address parameters. For example:

```
require(address != address(0), "Zero address not allowed");
```

Add these validation checks to: - constructors; - initialization functions; and - setter functions that update address parameters.

Acknowledgment 1.1

The issue was acknowledged with the following comment.

> *This is left offchain to the deployment scripts.*
>
> — Vfat

[Go back to Findings Summary](#)

I1: Duplicate code

Impact:	Info	Likelihood:	N/A
Target:	SickleRegistry.sol, FeesLib.sol	Type:	Code quality

Description

In the `ConnectorRegistry` contracts, the functions `connectorOf` and `hasConnector` have similar logic with the only difference in the return value. The algorithm of finding the connector address can be moved to a new internal function that returns the connector address if found, or zero address if not. The `connectorOf` and `hasConnector` functions can call this new function and return the correct value based on the output of this internal function.

Recommendation

Refactor the code to avoid code duplication.

Fix 1.1

The issue was fixed.

[Go back to Findings Summary]

I2: Usage of magic constants

Impact:	Info	Likelihood:	N/A
Target:	SickleRegistry.sol, FeesLib.sol	Type:	Code quality

Description

In the `SickleRegistry` contract, the `setFees` function verifies if the provided fee does not exceed the maximum value of 5%:

Listing 23. Excerpt from SickleRegistry

```
if (feesArray[i] <= 500) {
    // maximum fee of 5%
    feeRegistry[feeHashes[i]] = feesArray[i];
} else {
```

Also, the `FeesLib` contract uses inline `10_000`:

Listing 24. Excerpt from FeesLib

```
uint256 amountToCharge = feeBasis * fee / 10_000;
```

Best practices suggest that magic constants (like `500`) are discouraged and named constants should be used instead.

Recommendation

Add a new `MAX_FEE` constant and use it instead of plain `500`.

Fix 1.1

The issue was fixed.

Go back to Findings Summary

I3: Unconsolidated storage variable definitions

Impact:	Info	Likelihood:	N/A
Target:	SickleMultisig.sol	Type:	Code quality

Description

The `SickleMultisig` contract defines three storage variables on lines 66-68 and one storage variable, `_signers`, on line 253. The location of the latter is unobvious and may lead to incorrect assumptions about the storage layout of the contract.

Recommendation

Place the `_signers` variable definition together with other storage variables at the top of the contract.

Fix 1.1

The issue was fixed.

[Go back to Findings Summary]

I4: Redundant storage variable

Impact:	Info	Likelihood:	N/A
Target:	Automation.sol	Type:	Code quality

Description

In the `Automation` contract, the `approvedAutomators` array is used to store a list of all approved automators. Additionally, the `approvedAutomatorsLength` variable stores the length of this array. This variable is incremented in the `_setApprovedAutomator` function and decremented in the `revokeApprovedAutomator` function. Other than that, `approvedAutomatorsLength` is never used. Since the length of the array can be obtained by calling `approvedAutomators.length`, the `approvedAutomatorsLength` variable can be considered redundant and can be removed to save gas.

Also in the `setApprovedAutomator function there is a missing check, if the approvedAutomator` is already in the `approvedAutomators` array.

Recommendation

Remove the `approvedAutomatorsLength` variable and add a check in the `_setApprovedAutomator` function to prevent adding the same automator twice.

Fix 1.1

The issue was fixed.

[Go back to Findings Summary]

I5: Mapping `isCustomRegistry` is redundant

Impact:	Info	Likelihood:	N/A
Target:	ConnectorRegistry.sol	Type:	Code quality

Description

The `isCustomRegistry` mapping in the `ConnectorRegistry.sol` contract stores boolean values that are never read or utilized in the contract's logic. The mapping is updated in two scenarios:

1. Set to `true` when a new `CustomRegistry` is added via `addCustomRegistry` or `updateCustomRegistry` functions; and
2. Set to `false` when a `CustomRegistry` is updated via `updateCustomRegistry` function.

This mapping is redundant because:

- The active `CustomRegistry` data is already stored in the `customRegistries` mapping; and
- Historical records of removed registries are tracked through the `CustomRegistryRemoved` event.

Recommendation

Remove the `isCustomRegistry` mapping from the `ConnectorRegistry.sol` contract.

Fix 1.1

The issue was fixed. The `isCustomRegistry` mapping was removed from the `ConnectorRegistry.sol` contract. The new created `isCustomRegistry` function returns a boolean value indicating whether a registry exists in the `customRegistries` mapping.

[Go back to Findings Summary](#)

I6: Inconsistent function naming convention

Impact:	Info	Likelihood:	N/A
Target:	NftSettingsRegistry.sol, PositionSettingsRegistry.sol	Type:	Code quality

Description

The codebase uses a mix of `snake_case` and `camelCase` for function names, which violates Solidity's style guide. Solidity's convention recommends using `camelCase` for function names.

The `NftSettingsRegistry.sol` contract contains the following functions using `snake_case` naming convention:

- `_get_sickle_by_owner`;
- `_set_nft_settings`;
- `_unset_nft_settings`;
- `_check_rebalance_config`; and
- `_check_tick_width`.

The `PositionSettingsRegistry.sol` contract contains the following functions using `snake_case` naming convention:

- `_check_reward_config`;
- `_get_sickle_by_owner`; and
- `_get_pool_price`.

Recommendation

Rename all functions to follow the `camelCase` naming convention according to the Solidity style guide. For example:

- `_get_sickle_by_owner` should be `_getSickleByOwner`;
- `_set_nft_settings` should be `_setNftSettings`; and
- `_check_reward_config` should be `_checkRewardConfig`.

Fix 1.1

The issue was fixed. The functions were renamed to follow the `camelCase` naming convention.

[Go back to Findings Summary]

I7: Typographical error in function comment

Impact:	Info	Likelihood:	N/A
Target:	NftSettingsRegistry.sol	Type:	Code quality

Description

The `NftSettingsRegistry.sol` contract contains a typographical error in the comment of the `_check_rebalance_config` function. The comment:

```
// Check configuratgion parameters for errors
```

contains a misspelling of the word ”configuration”.

Recommendation

Correct the comment to:

```
// Check configuration parameters for errors
```

Fix 1.1

The issue was fixed. The comment was corrected to ”Check configuration parameters for errors”.

Go back to Findings Summary

I8: Misleading error name

Impact:	Info	Likelihood:	N/A
Target:	SickleMultisig.sol	Type:	Code quality

Description

The `TransactionNotReadyToExecute` error in the `SickleMultisig.sol` contract is used when a transaction fails due to insufficient signature count. The error name is misleading because:

- It does not explicitly indicate that the failure is due to insufficient signatures; and
- The term ”not ready” is ambiguous and could suggest other potential issues.

Listing 25. Excerpt from SickleMultisig

```
if (transaction.signatures < threshold) {
    revert TransactionNotReadyToExecute();
}
```

Recommendation

Rename the error to clearly indicate the specific reason for the transaction failure.

Fix 1.1

The issue was fixed. The error name was changed to `InsufficientSignatures` to clearly indicate the specific reason for the transaction failure.

Go back to Findings Summary

I9: Unused errors

Impact:	Info	Likelihood:	N/A
Target:	NftSettingsRegistry.sol	Type:	Code quality

Description

The contract has unused errors. The following code excerpts enumerate all of them.

Listing 26. Excerpt from NftSettingsRegistry

```
error CompoundOrHarvestNotSet();
error CompoundAndHarvestBothSet();
```

Recommendation

Remove the unused errors or utilize them.

Fix 1.1

The issue was fixed. The unused errors were removed from the contract.

Go back to Findings Summary

I10: Redundant function

Impact:	Info	Likelihood:	N/A
Target:	Sickle.sol	Type:	Code quality

Description

The `_Sickle_initialize` function in the `Sickle.sol` contract serves only as a pass-through to `SickleStorage._SickleStorage_initialize` without adding any additional functionality:

Listing 27. Excerpt from Sickle

```
function _Sickle_initialize(
    address sickleOwner_,
    address approved_
) internal {
    SickleStorage._SickleStorage_initialize(sickleOwner_, approved_);
}
```

The function is called in the constructor and `initialize` function of the `Sickle.sol` contract:

Listing 28. Excerpt from Sickle

```
_Sickle_initialize(address(0), address(0));
```

Recommendation

Remove the redundant `_Sickle_initialize` function and call `_SickleStorage_initialize` directly in both the constructor and `initialize` function.

Fix 1.1

The issue was fixed. The redundant `_Sickle_initialize` function was removed and `_SickleStorage_initialize` was called directly from the `initialize` function. The call in the constructor was removed.

Go back to Findings Summary

I11: Missing duplicate registry validation

Impact:	Info	Likelihood:	N/A
Target:	ConnectorRegistry.sol	Type:	Data validation

Description

The `ConnectorRegistry` contract lacks validation checks for duplicate registries in the `addCustomRegistry` and `updateCustomRegistry` functions. The same registry address can be added multiple times to the `customRegistries` array, leading to redundant entries and increased gas costs for array operations.

Recommendation

Implement validation checks in both functions to prevent duplicate registry addresses.

Partial solution 1.1

The issue was partially fixed. The duplicate registry validation was added to the `addCustomRegistry` function. The `updateCustomRegistry` function was not updated.

[Go back to Findings Summary]

I12: Errors in documentation

Impact:	Info	Likelihood:	N/A
Target:	SickleStorage.sol	Type:	N/A

Description

Misleading documentation for the `SickleStorage.onlyOwner` modifier. The statement ”if the admin was not set yet, the modifier will not restrict the call” is not true.

Listing 29. Excerpt from SickleStorage

```
/// @dev Restricts a function call to the owner, however if the admin was
/// not set yet,
/// the modifier will not restrict the call, this allows the SickleFactory
/// to perform
/// some calls on the user's behalf before passing the admin rights to them
modifier onlyOwner() {
    if (msg.sender != owner) revert NotOwnerError();
    _;
}
```

Recommendation

Fix the documentation or implementation.

Fix 1.1

The issue was fixed.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Member's Name	Position
Štěpán Šonský	Lead Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Vfat: Sickle, 9.5.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz