PUBLIC

Code Assessment

of the Aerodrome Module Smart Contracts

January 27, 2025

Produced for

arrakis

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Arrakis Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Aerodrome Module according to Scope to support you in forming an opinion on their security risks.

Arrakis Finance implements a module for Arrakis Modular that integrates with the Concentrated Liquidity Pools and reward mechanisms of Aerodrome on the Base blockchain. The module is used in Arrakis' private modules which are not open to public deposits.

The most critical subjects covered in our audit are functional correctness, integration with the rest of the Arrakis system, integration with Aerodrome and asset solvency. Security regarding the mentioned subjects is good. Most issues have been fixed, but some small problems remain.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	8
• Code Corrected	3
• Code Partially Corrected	2
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Aerodrome Module repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	03 January 2025	24620f3bc5e9c18af4355dfe1439a73f0a3ba174	Initial Version
2	20 January 2025	7005cc6bbd2dbb28130a03ebd66c52e78f1ebaad	After Intermediate report

For the solidity smart contracts, the compiler version `0.8.26` was chosen.

The following files were in scope:

```
src/modules/AerodromeStandardModulePrivate.sol
```

2.1.1 *Excluded from scope*

All other files are not in scope. The core has been part of another review. Further, the correctness of Aerodrome is out of scope. Aerodrome is expected to function as documented. Incompatible tokens are out of scope. Please refer to our core audit report for further details involving assumptions made by the core.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Arrakis Finance offers an Arrakis Modular module integrating with Concentrated Liquidity Pools (Slipstream) of Aerodrome. The module is used in private Arrakis meta vaults only. This overview describes the new module. For details regarding the protocol's core, refer to the core audit report.

A set of privileged depositors can add funds to the `AerodromeStandardModulePrivate` contract via an associated private metavault. Another set of privileged addresses, the executors, can then proceed to provide liquidity to any Aerodrome Concentrated Liquidity `CLPool` that is composed of the given tokens (chosen by the vault owner). The resulting NFT describing the liquidity position is, thereby, automatically staked in the appropriate `CLGauge` contract, thus earning AERO rewards based on the voting results of Aerodrome's `Voter` contract. AERO rewards are accumulated in the module on every interaction with the positions.

The vault owner can withdraw their share of AERO rewards and proceed to manually lock them in Aerodrome's `VotingEscrow` contract, where they can earn trading fees from the pools. This step is, contrary to other modules which directly accrue fees and add them to the module's managed funds, not part of the module's workflow and entirely up to the vault owner. Similarly, Arrakis' share of the fees is not released like in other modules but via a special function.

`AerodromeStandardModulePrivate` implements the following functions:

1. `fund()`: Whitelisted depositors can transfer any amount of the two tokens of the vault to the module. The tokens are not immediately invested in the pool, rather they are left "idle" in the module, waiting for a `rebalance()` call to explicitly allocate them to a position.
2. `withdraw()`: The vault owner can withdraw a share of the available funds to any address.
3. `claimRewards()`: The vault owner can transfer all AERO rewards (apart from the share that is owed to Arrakis) to any address.
4. `claimManager()`: Anyone can send AERO rewards owed to Arrakis to the address set by `setReceiver()`.
5. `rebalance()`: Executors can shuffle all funds between different positions on a given Aerodrome pool. They can also perform arbitrary swaps.
6. `withdrawManagerBalance()`: Unused.
7. `approve()`: The metavault owner can create approvals for any idle funds that have not been sent to a pool (yet). This can be used to enable swaps with delayed execution (e.g., based on signed transaction execution).

Rebalancing is subject to slippage protection such that (partially trusted) executors are not able to extract a huge chunk of the funds in one go. For that, the module contains the functions `totalUnderlying()`, which computes the token amounts the module is currently worth, and `validateRebalance()`, which uses the price of an external oracle to make sure that the pool has not been imbalanced before or during execution.

Furthermore, the module can be paused/unpaused by a certain pauser that is defined in the `Guardian` contract. This disables all functionality besides withdrawals.

2.2.1 Roles and Trust Model

The module defines the following roles:

- Meta Vault Owner: Fully trusted. The owner can drain all funds by creating arbitrary approvals for the held tokens.
- Metavault: Fully trusted and expected to be the `ArrakisMetaVaultPrivate` contract. In case the implementation is changed, the module can be drained (by withdrawing all funds).
- Manager: Fully trusted and expected to be the `ArrakisStandardManager` contract. In case the implementation is changed, the module can be drained (by rebalancing without slippage protection).
- Guardian: Partially trusted to pause and unpause responsibly.
- Executors: Partially trusted. The executors cannot instantaneously drain the protocol. However, adversarial executors might drain a vault slowly. See Possibilities of executors to drain funds for details.
- Users: Fully untrusted.

Note that the system defines several other roles (e.g. proxy owners). See the core audit report for more details.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5

- Inconsequential Minimum-Returns Check Code Partially Corrected
- Missing Slippage Protection Risk Accepted
- Token Allowance Abuse During Module Change Risk Accepted
- Token Allowance Abuse During Rebalance Risk Accepted
- Vault Owner Can Steal Manager Fees Code Partially Corrected Risk Accepted

5.1 Inconsequential Minimum-Returns Check

Correctness Low Version 1 Code Partially Corrected

CS-ARRKS-MOD-AD-003

The internal functions `_decreaseLiquidity()` and `_increaseLiquidity()` pass non-zero `amount0Min` and `amount1Min` to the corresponding functions of Aerodrome's `NonFungiblePositionManager`, in order to protect from spot-price manipulation. These amounts are computed as proportions of the amounts virtually sitting in the position at a reference spot price, supplied as a parameter. However, the callers of these functions, namely `rebalance()` and `withdraw()`, take this spot price from the current (possibly manipulated) state of the pool, rather than from a trusted oracle. As a result, the effective protection vanishes, since the minimum amounts will be computed from the actual amounts currently sitting in the position, at whatever (possibly manipulated) spot price the pool is currently at: In other words, the protective checks will never revert.
Incidentally, the proportion is also computed incorrectly: A numerator of `_maxSlippage`, instead of `PIPS - _maxSlippage`, is supplied to `FullMath.mulDiv()`.

Code partially corrected:

The `rebalance()` function now fetches an oracle price to use with the functions `_decreaseLiquidity()` and `_increaseLiquidity()`.

Nevertheless:

1. The withdraw function still uses the spot price. This seems, however, intentional as the risk of Missing Slippage Protection has been accepted.

2. `_decreaseLiquidity()` no longer computes the proportion of `amt0` / `amt1` resulting in slippage checks with possibly too large values.
3. The calculation of the actual slippage values is still incorrect (`amount * maxSlippage` instead of `amount * (1 - maxSlippage)`) resulting in slippage checks with possibly too small values.

5.2 Missing Slippage Protection

Design Low Version 1 Risk Accepted

CS-ARRKS-MOD-AD-005

`AerodromeStandardModulePrivate.withdraw()` is always called directly on the vault since `ArrakisPrivateVaultRouter` does not offer wrapper functions for removing liquidity. Since the `withdraw()` function does not have any slippage checks, slippage is possible: If the underlying pool is imbalanced before the withdrawal, a different token ratio than anticipated will be withdrawn. The vault owner withdrawing tokens might receive token amounts not anticipated when they created the transaction.

Risk accepted:

Arrakis Finance accepts the risk.

5.3 Token Allowance Abuse During Module Change

Security Low Version 1 Risk Accepted

CS-ARRKS-MOD-AD-008

Executors can call arbitrary payloads (except the `withdraw()` function) on any new module that is set with `ArrakisMetaVault.setModule()`. A call to a module's fund function can be used to abuse previously set token allowances by users of the given module.

The function `AerodromeStandardModulePrivate.fund()` accepts the address of the depositor as arguments and then proceeds to transfer ERC-20 tokens from this address to the module using `transferFrom()`. Any existing allowances to the module can be used to transfer larger amounts than originally planned by the depositor.

Executors can wait until enough allowances have been set to the module, switch to another module and back to the given module, executing `fund()` payloads for each depositor with an open allowance.

Note that the severity is given based on the assumption that depositors will interact with the router contract and not directly with the module.

Risk accepted:

Arrakis Finance states:

```
Any user or protocol interacting with vault or their modules directly do it on their own risk.
A safe interaction with the vault/modules should happen through routers.
```

Note that it is of utmost importance that integrating protocols do not give approval to the respective contracts.

5.4 Token Allowance Abuse During Rebalance

Security Low Version 1 Risk Accepted

CS-ARRKS-MOD-AD-009

`AerodromeStandardModulePrivate.rebalance()` allows executors to perform arbitrary calls from the module. The calls must return a certain minimum of one of the vault's tokens but have no other restrictions. The executor could call one of the token contracts and transfer funds from any depositor that has an open allowance to the module.

Since funds are transferred, the swap's `amountIn` and `expectedMinReturn` can be chosen in a way that ensures execution. The slippage check in `ArrakisStandardManager.rebalance()`, however, prevents the executor from adding too many funds this way as it also reverts on positive slippage.

Note that the severity is given based on the assumption that end users will interact with the router contract.

Risk accepted:

Arrakis Finance states:

```
We are assuming that only interacting through the router is safe. Protocols integrating Arrakis
Modular should also use the router.
```

5.5 Vault Owner Can Steal Manager Fees

Security Low Version 1 Code Partially Corrected Risk Accepted

CS-ARRKS-MOD-AD-007

Assuming the owner of a private vault fully controls one of the vault's tokens (say `token1`) and has, e.g., the capability to upgrade it to any implementation with zero delay, this allows the slippage check around the opaque swap call in `AerodromeStandardModulePrivate.rebalance()` (which requires at least 1 wei of token to be sent back to the module) to be bypassed by simply having the second call to `token1.balanceOf()` return an artificially-inflated value.
Together with the absence of extensive sanity checks on the "swap router" address, this allows the vault executor to initiate a `rebalance()` containing a "swap" that actually encodes a call to `Gauge.withdraw()`. This sends all of the position's AERO rewards back to the module, without cutting the due fee for the manager and accruing it into `_aeroManagerBalance`. This effectively allows the vault owner to steal the manager fees, while satisfying the slippage check (`totalUnderlying()` still counts tokens' liquidity even if they are not held by the gauge).

Code partially corrected:

The gauge can no longer be called during the swap call.

Risk accepted:

There is still a remaining risk that `AERO.transfer()` is called to transfer all held AERO tokens to another address. Arrakis Finance accepts this risk.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- AERO Manager Balance Is Not Reset Code Corrected

Low-Severity Findings	3

- Executor Can Set the AERO Receiver Code Corrected
- Missing Checks Code Corrected
- Vault Owner Can Gain Approval for Module's NFT Positions Code Corrected

6.1 AERO Manager Balance Is Not Reset

Correctness Medium Version 1 Code Corrected

CS-ARRKS-MOD-AD-001

The function `AerodromeStandardModulePrivate.claimManager()` transfers the due `_aeroManagerBalance` to the manager without resetting it to 0 afterwards. Therefore, this amount keeps increasing indefinitely, leading to an excessive manager fee payout starting from the second claim.

Code corrected:

`_aeroManagerBalance` is now reset to zero before the tokens are transferred.

6.2 Executor Can Set the AERO Receiver

Design Low Version 1 Code Corrected

CS-ARRKS-MOD-AD-002

The function `AerodromeStandardModulePrivate.setReceiver()`, having the `onlyManager` modifier but not being part of the canonical module interface, can only be called through the `rebalance()` function of the `ArrakisStandardManager`. This function, in turn, can only be called by an "executor" chosen by the owner of the (private) vault: This executor is thus not the same entity as the Arrakis management, and may act adversarially towards it. In particular, they could set the module's AERO receiver to an address they control in order to steal the manager fees.

Code corrected:

The AERO receiver is now set by the owner of the `ArrakisStandardManager`.

6.3 Missing Checks

Design Low Version 1 Code Corrected

CS-ARRKS-MOD-AD-004

The following checks are missing:

1. `AerodromeStandardModulePrivate.initialize()` does not check, whether one of the vault tokens is the AERO token. In this case, rewards and funds would be commingled, allowing executors to create positions using funds that belong to the Arrakis Finance manager.
2. `AerodromeStandardModulePrivate.initialize()` does not check, whether the gauge of the respective pool is valid or alive.

Code corrected:

Both checks have been added.

6.4 Vault Owner Can Gain Approval for Module's NFT Positions

Security Low Version 1 Code Corrected

CS-ARRKS-MOD-AD-006

The executor (controlled by the vault owner, and thus possibly adverse to the Arrakis management) supplies an arbitrary "router" address and an arbitrary opaque "swap payload", that the module will use in a low-level `call`, to the `AerodromeStandardModulePrivate.rebalance()` function. The combined effects of the pre-swap `_checkMinReturn()` function and the post-swap minimum returns check mean that the overall rebalance succeeds only if some positive amount of either `token0` or `token1` flows into the module as a result of the swap.
The executor can pre-mint a dummy NFT position, with little funds inside, through Aerodrome's `NonFungiblePositionManager` contract and donate it to the module. Then, since the NFT contract is `Multicall`, it can choose the "router" address to be the NFT contract address and a payload encoding a `multicall()` to:

1. `setApprovalForAll(attacker)`, to gain permanent approval for all of the module's NFT positions.
2. `decreaseLiquidity(tokenId, allLiquidity)`, which will move the position's funds into its `tokensOwed` fields.
3. `collect(tokenId)`, which will actually transfer the token amounts back to the module, ensuring that the slippage check succeeds.

There is currently no way for an executor to actually exploit this permanent approval, but this could change if, for example, the Arrakis management were to upgrade the implementation address in the `UpgradableBeacon` responsible for the Aerodrome module and point it to a new implementation that does not always stake all the NFT positions into the Aerodrome gauge. In that case, the approval would persist for the same address, and the attacker could transfer those unstaked NFTs to themselves and directly call `collect()` on them, effectively stealing the manager fees.

Code corrected:

The `NonFungiblePositionManager` (as well as the gauge) can no longer be called during the swap call in `rebalance()`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Shadowed Variable

Informational Version 1

CS-ARRKS-MOD-AD-011

`AerodromeStandardModulePrivate._increaseLiquidity()` defines a variable `gauge` which shadows a storage variable with the same name. This is generally considered bad practice and can potentially lead to erroneous behavior during code maintenance.

7.2 Miscellaneous Remarks

Informational Version 1 Acknowledged

CS-ARRKS-MOD-AD-010

The following missing checks could introduce problems during contract maintenance (internal or external):

1. `AerodromeStandardModulePrivate.initializePosition()` can be called by anyone. This is currently unproblematic as the function does not contain any logic.
2. `AerodromeStandardModulePrivate._mint()` does not check the receiver of the NFT before it is passed to the `NonfungiblePositionManager`. This is currently unproblematic as the later deposit to the gauge would fail if the receiver was set to anything other than the contract itself.

Acknowledged:

Arrakis Finance acknowledges the remarks.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Approval Considerations

Note Version 1

`AerodromeStandardModulePrivate.approve()` allows the metavault owner to give approvals to arbitrary addresses.

As a consequence, the feature could be used to drain funds. However, note that as part of the trust model, the owner is fully trusted. It is thus expected that only trusted address will be approved.

Nevertheless, below is a non-exhaustive list of considerations for owners when approving addresses:

- The approved address is fully trusted. In case of issues, the module could be attacked (limited by the approval amounts). Optimally, the approved address is a contract with limited but clearly defined functionality.
- No asynchronous value exchanges are allowed. Namely, if some approval is used, tokens with an equivalent value shall be returned to the module immediately. Note that there are no slippage checks present in the module.

Ultimately, approved addresses must be evaluated carefully to ensure the safety of users' funds.

8.2 Possibilities of Executors to Drain Funds

Note Version 1

Executors can call arbitrary functions on arbitrary contracts during `rebalance()` to be able to swap tokens. These calls are constrained by the fact that `amountIn` and `expectedMinReturn` have to be set to some value other than 0, which requires the call to return at least 1 wei of one of the vault's tokens back to the module. Additionally, rebalancing must satisfy the constraints by `validateRebalance()` and the slippage check against `totalUnderlying()`. Furthermore, rebalancing can only be performed after a given cooldown period. All of these constraints, however, still allow for the extraction of a small part of the funds. Consider the following examples:

1. The executor calls their own contract which transfers a specific `amountIn` token0 (or token1) from the module and sends back the least amount of token1 (or token0) still covered by `_checkMinReturn()`.
2. The executor performs a normal swap an some AMM but sandwiches the whole rebalance call so that they can siphon off the maximum slippage.

It is also worth noting that the first point can be repeated multiple times (as `ArrakisStandardManager.rebalance()` allows calling the module's rebalance function multiple times in a row) if the slippage protection for the swap in the module is set tighter than the slippage protection in the standard manager.