Code Assessment

of the Uniswap V4 Module Smart Contracts

December 13, 2024

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 for the Uniswap V4 Module of Arrakis Modular according to Scope to support you in forming an opinion on their security risks.

Arrakis Finance implements modules integrating with Uniswap V4 for Arrakis Modular. That allows managers to manage a vault's liquidity on Uniswap V4.

The most critical subjects covered in our audit are functional correctness, integration with Arrakis Modular and external systems, asset solvency and precision of arithmetic operations. The general subjects covered are specification, gas efficiency, and trustworthiness.

The most significant findings are:

- Array manipulation during iteration
- Bad rounding
- Manager fee collected multiple times
- Token allowance abuse during module change

The first three items have been corrected through code corrections while the risk for the last one has been accepted. Note that other lower severity issues have been partially corrected or acknowledged.

It is also worth noting that the project is subject to certain roles that are not fully trusted and can, theoretically, extract small parts of the liquidity in discrete timer intervals. See Possibilities of executors to drain funds for details.

In summary, we find that the codebase provides a good level of security, although it depends on the correct usage by trusted accounts.

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	3
• `Code Corrected`	2
• `Risk Accepted`	1
`Low`-Severity Findings	18
• `Code Corrected`	14
• `Code Partially Corrected`	2
• `Risk Accepted`	1
• `Acknowledged`	1

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 Arrakis Modular 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	18 Oct 2024	5d6c5f7f795fd0732d137ac189dd5967bef858e3	Initial Version
2	25 Nov 2024	abc0a63e236a3f03271b17ba26685ca82c1d604d	After Intermediate Report
3	29 Nov 2024	601b7bda3c2c20ef898a0ca8dfa34a28923a953d	Final Version
4	13 Dec 2024	793588ed55d7e2d36fd749fdde406a606ff7f8f9	Refactor and Partial Fix
5	13 Dec 2024	503682751808527a373a767ecdf639bc2626a909	Final Version

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

The files in scope are:

```
src/
    abstracts/UniV4StandardModule.sol
    hooks/ArrakisPrivateHook.sol
    interfaces/
        IUniV4ModuleBase.sol
        IUniV4StandardModule.sol
        IArrakisPrivateHook.sol
        IUniV4StandardModuleResolver.sol
    libraries/UnderlyingV4.sol
    modules/
        UniV4StandardModulePublic.sol
        UniV4StandardModulePrivate.sol
        resolvers/UniV4StandardModuleResolver
    structs/SUniswapV4.sol
```

In Version 4, the following contract was added as part of a refactoring:

```
src/libraries/UniswapV4.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 Uniswap is out of scope. Uniswap 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.

Note that Uniswap v4 is not deployed yet and may be subject to changes. The commits used for the Uniswap v4 codebase are the relevant submodule commits of the initially reviewed version.

2.2 System Overview

This system overview describes the initially received version ($\boxed{\text{Version 1}}$) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

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 implements an Arrakis Modular module integrating with Uniswap v4, allowing the protocol to manage Uniswap v4 liquidity. A public and a private version are provided to support the respective metavaults. The overview describes the new module. For details regarding the protocol's core, refer to the core audit report.

The module allows managers to manage positions on one Uniswap v4 pool at a time. Hence, funds are allocated to tick ranges of a pool. Note that idle funds are wrapped to ERC-6909 balances. Similarly, the fees earned on Uniswap v4 will be wrapped to ERC-6909 balances until they are rebalanced into new or existing positions. On each interaction, the manager's share of the fees is sent to another contract (the standard manager) from which it can be distributed to the appropriate receiving addresses.

The following functions for adding and removing liquidity are provided (used by the vault):

1. `deposit()` (only public module): Provides liquidity proportionally to each tick range and the ERC-6909 balances. Fees are collected accordingly.

2. `fund()` (only private module): Provides liquidity arbitrarily to ERC-6909 balances (so that the balances can be used later in rebalancing).

3. `withdraw()`: Withdraws liquidity proportionally from each tick range and from the ERC-6909 balances (similar to `deposit()`). Note that a full withdrawal (as required in `setModule()`) is supported. Further, fees will be collected accordingly.

4. `initializePosition()`: Wraps the full token balances received from another module during a module migration defined in `setModule()` to the respective ERC-6909 tokens.

The following functions for managing the module's liquidity are provided (usable by the standard manager through its `rebalance()` function):

1. `rebalance()`: A rebalancing initially applies liquidity deltas to tick ranges to add and remove liquidity for positive and negative deltas, respectively. While in both cases fees will be collected, specifying the delta to be zero allows collecting fees only. Note that this tracks and untracks tick ranges accordingly (tick ranges with non-zero liquidity are tracked). Next, an optional swap can be performed through an arbitrary router contract. Finally, all deltas with Uniswap's pool manager are settled. Note that this includes token transfers (for the flashloaned amount and received amount from the swap) and ERC-6909 minting / burning.

2. `setPool()`: Changes the active pool and rebalances the funds on the new pool. For the old pool, a rebalancing (as in `rebalance()`) without a swap is performed that removes all liquidity. Then, for the new pool, a regular rebalancing is performed.

Note that `view` functions required for rebalancing are implemented as follows:

1. `totalUnderlying()`: Sums up the tokens currently present (according to the current tick) in the active tick ranges, the tokens wrapped in the ERC-6909 balances, and the users' share of the yet-to-be-claimed LP fees.

2. `validateRebalance()`: Validates the pool's price against the standard manager's oracle price. Reverts if the deviation is too high. Note that this prevents creating positions at bad prices that would allow draining the protocol.

Further, `withdrawManagerBalance()`, callable by arbitrary addresses but typically called by the metavault or the standard manager, is implemented to allow claiming all generated fees and withdrawing the manager's share. Note that this collects fees on all tick ranges (as in `rebalance()` when operating on all active tick ranges) but without a swap. Additionally, `setManagerFeePIPS()` is implemented to allow the standard manager to set its fee. Last, `pause()` and `unpause()` allow the guardian to pause all functionality.

Arrakis Finance offers a dedicated Uniswap v4 Hook that can be used with any Uniswap v4 pool. It allows only a pre-defined address to add liquidity and defines two different fee tiers depending on whether users swap from token0 to token1 or token1 to token0.

2.2.1 Changes in Version 2

The following changes (besides corrections) were applied in the second version of the codebase:

1. The locally held balances are not held as the wrapped ERC-6909 tokens on Uniswap but as the underlying ERC-20 tokens directly on the module.

2. The meta vault's owner can now approve arbitrary addresses with `approve()`. These contracts can then pull the tokens from the module. Note that for ETH, a local approval can be given so that approved address can call the newly added function `withdrawEth()` to retrieve the native token. This is done to enable swaps that are performed at a later time (e.g., based on signed transaction execution).

2.2.2 Changes in Version 4

Much of the logic was moved to the public library `UniswapV4` as part of refactoring.

2.2.3 Roles and Trust Model

The module defines the following roles:

- Guardian: Trusted to pause and unpause responsibly.

- Metavault: Fully trusted and expected to be the private or public version accordingly. In case another vault implementation would be set, the module could be drained.

- Manager: Fully trusted and expected to be the standard manager contract. In case another implementation would be set, the module could be drained.

- Executors in Manager: 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.

- Meta Vault Owner: The owner of the meta vault has received additional privileges in `Version 2`. Namely, the owner can now drain all funds by approving all tokens with the newly added functionality described in Changes in Version 2.

Note that the system defines several other roles (e.g. vault 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	1

- Token Allowance Abuse During Module Change `Risk Accepted`

`Low`-Severity Findings	4

- Deposit Frontrunning `Code Partially Corrected` `Risk Accepted`
- Missing Cooldown Period Minimum `Acknowledged`
- Token Allowance Abuse During Rebalance `Risk Accepted`
- Token Donations `Code Partially Corrected` `Risk Accepted`

5.1 Token Allowance Abuse During Module Change

`Security` `Medium` `Version 1` `Risk Accepted`

CS-ARRKS-MOD-UNIV4-003

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 deposit function can be used to abuse previously set token allowances by users of the given module.

The functions `UniV4StandardModulePublic.deposit()` and `UniV4StandardModulePrivate.fund()` accept the address of the depositor as arguments and then proceed to transfer ERC-20 tokens from these addresses to the Uniswap `PoolManager` using `transferFrom()`. Any existing allowances can be used to create new positions without any vault shares being minted (as the vault's `mint()` function is bypassed). Thus, the new positions benefit all existing depositors.

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

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

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.

5.2 Deposit Frontrunning

`Security` `Low` `Version 1` `Code Partially Corrected` `Risk Accepted`

CS-ARRKS-MOD-UNIV4-006

Executors can get free shares on empty vaults. They can use this to frontrun the first deposit on a vault and thus gain parts of the deposit. The scheme works as follows:

1. A new vault with the `UniV4StandardModulePublic` is deployed.

2. Another, arbitrary module is whitelisted on the vault.

3. 1e18 shares are minted through `ArrakisMetaVault.mint()`.

4. The executor frontruns this deposit:

   1. 1 wei of one of the vault tokens is donated to the module. This resets the `init0` / `init1` functionality.

   2. 1e18 shares are minted by depositing 1 wei of the previously donated tokens.

   3. `ArrakisStandardManager.rebalance()` is called with two ranges that are chosen such that the token amounts for the created liquidity are rounded up to 1 wei.

   4. `ArrakisStandardManager.setModule()` is called to change the module. The withdrawal of the two ranges results in 0 tokens withdrawn (due to the planned rounding error). The new module is now initialized with no tokens.

   5. `setModule()` is be called again to switch back to the `UniV4StandardModulePublic`.

   6. Since the module does not contain any value, the init values are active again. The frontrun deposit now deposits a proportion of 1e18 - exactly the init amounts.

   7. The depositor only owns half of the deposited tokens. The other tokens are now owned by the executor.

Please note that this attack is only viable if done directly through the vault and not through the router. Furthermore, the first deposit must be done after the deployment in a non-atomic way.

Code partially corrected:

While the `init0` / `init1` functionality can no longer be reset by simple donations, the executor still has the means to perform the attack. If a new vault is deployed with two modules, switching to another module, donating some wei to the old module and then switching back will set `notFirstDeposit` in the `UniV4StandardModulePublic` to `true` in `initializePosition()`. The executor can then proceed to donate underlying tokens to the module and carry on with the attack as described.

Risk accepted:

Arrakis Finance states that the remaining attack vector will be mitigated by the team performing a small mint directly after deployment of a new vault, ensuring that the first deposit can not be performed by an executor.

5.3 Missing Cooldown Period Minimum

Design   Low   Version 1   Acknowledged

CS-ARRKS-MOD-UNIV4-009

Executors can call `ArrakisStandardManager.rebalance()` in order to run the `rebalance()` function on the `UniV4StandardModule`. This allows them to move funds between different positions on the Uniswap `PoolManager`. Since required token ratios differ between positions with different ticks, executors can choose to execute a swap during rebalancing. Such swaps can incur losses due to slippage (which could be absorbed by the executors themselves) and thus, a maximum slippage percentage is set in the module (and another one in the `ArrakisStandardManager`) to ensure that executors can only incur a certain amount of slippage at a time.

To make sure that executors cannot drain the vaults, another mechanism is used that allows them to only call `rebalance()` in discrete time intervals. However, this cooldown period can be set to any value greater than 0. While this prevents draining the contract completely in one block, low values would allow executors to drain the maximum available slippage multiple times in short time periods.

Acknowledged:

Arrakis Finance stated that it is assumed that the owners should choose cooldown values responsibly.

5.4 Token Allowance Abuse During Rebalance

Security   Low   Version 1   Risk Accepted

CS-ARRKS-MOD-UNIV4-015

`UniV4StandardModule._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 user that has an open allowance to the module. These funds are then settled with the `PoolManager`, creating a new delta for which new ERC-6909 tokens are minted later. These new funds benefit all existing holders.

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 Token Donations

Design   Low   Version 1   Code Partially Corrected   Risk Accepted

CS-ARRKS-MOD-UNIV4-016

`UniV4StandardModule` defines the values `_init0` and `_init1` that specify the token amount ratios for the first deposit. This is enforced with the following code:

```
currency0Id = CurrencyLibrary.toId(poolKey.currency0);
leftOver0 = IERC6909Claims(address(poolManager)).balanceOf(
    address(this), currency0Id
);

currency1Id = CurrencyLibrary.toId(poolKey.currency1);
leftOver1 = IERC6909Claims(address(poolManager)).balanceOf(
    address(this), currency1Id
);

...

if (length == 0 && leftOver0 == 0 && leftOver1 == 0) {
    leftOver0 = _init0;
    leftOver1 = _init1;
}
```

However, `PoolManager` balances are transferrable. This means that anyone can donate some of these tokens to the module beforehand to disable the mechanism and therefore set the ratio themselves.

Code partially corrected:

Regular token donations no longer reset the `_init0` / `_init1` ratio. Executors, however, are still able to circumvent the ratio in the following circumstances:

- A new public vault has been deployed.
- A second module has been added to the vault.

In this case, executors can call `setModule()` two times to automatically set `notFirstDeposit` to `true` (they have to donate some tokens to the module after thet first `setModule()` call), circumventing the new mechanism for initial deposits.

Risk accepted:

Arrakis Finance states that the remaining attack vector will be mitigated by the team performing a small mint directly after deployment of a new vault, ensuring that the first deposit can not be performed by an executor.

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	2

- Array Manipulation During Iteration `Code Corrected`
- Manager Fee Collected Multiple Times `Code Corrected`

`Low`-Severity Findings	14

- Allowance Abuse Through Hooks `Code Corrected`
- Bad Rounding `Code Corrected`
- Lack of Event Emissions `Code Corrected`
- Manager Fee Finalization DoS `Code Corrected`
- Missing Init Minima `Code Corrected`
- Missing Range Check `Code Corrected`
- No Accrual During Fee Change `Code Corrected`
- Pausing Without Effects `Code Corrected`
- Private Hook Unusable `Code Corrected`
- Uni V4 Resolver Ignores Token Order `Code Corrected`
- Uni V4 Resolver Math `Code Corrected`
- Wrong Counting of Minted and Burned Tokens `Code Corrected`
- Wrong Liquidity Invariant `Code Corrected`
- UnderlyingV4 Ignores Management Fees for Mint Amounts `Code Corrected`

Informational Findings	5

- Incorrect Unused Leftovers `Code Corrected`
- Balance Function Errors `Code Corrected`
- Empty Return Values `Code Corrected`
- Irrelevant Actual Balances in Rebalance() `Code Corrected`
- NatSpec Inaccuracies `Code Corrected`

6.1 Array Manipulation During Iteration

`Correctness` `Medium` `Version 1` `Code Corrected`

CS-ARRKS-MOD-UNIV4-001

`UniV4StandardModule._withdraw()` iterates over the `_ranges` array to withdraw an equal amount from all positions. If the full liquidity of a position is withdrawn, its corresponding range is removed from the `_ranges` array:

```
uint256 length = _ranges.length;

for (uint256 i; i < length; i++) {
    Range memory range = _ranges[i];
    ...
    if (liquidity == uint256(state.liquidity)) {
        ...
        _ranges[indexToRemove] = _ranges[l - 1];
        _ranges.pop();
    }
    ...
}
```

The array length is decreased during this operation, leading to an out-of-bounds access once the iteration progresses to the end of the now shortened array.

Code corrected:

Now, the iteration is performed on a memory array. All removals are performed on the storage array.

Note that in version 4, an issue has been introduced as part of a gas optimization but has been resolved in version 5.

6.2 Manager Fee Collected Multiple Times

`Correctness` `Medium` `Version 1` `Code Corrected`

CS-ARRKS-MOD-UNIV4-002

Every time a position in the Uniswap `PoolManager` is updated, accrued fees are added to the unlock delta. In this case, the `UniV4StandardModule` calculates the manager fee, transfers it to the `ArrakisStandardManager` and mints ERC-6909 tokens for the remaining delta.

During (public vault) deposits and rebalances, the accrued fees for all positions are first calculated before the positions are updated.

`UniV4StandardModulePublic._deposit()` iterates through all available positions and calls `PoolManager.modifyLiquidity()`. In contrast, that is not the case for `UniV4StandardModule._rebalance()` where the function only iterates over the positions supplied by the executor. Meaning, that the function first calculates the manager fee on all positions but then possibly only calls `modifyLiquidity()` on some of them. The remaining positions will yield the same fees that have already been accounted for the next time they are updated in the `PoolManager`. Consider the following example:

1. The manager fee is 10%.

2. The module holds 2 positions #0 and #1.

3. After some time, both positions have accrued 10 tokens in fees each.

4. An executor calls `rebalance()`, reducing position #0 and creating a new position #2.

5. 2 tokens are sent to the manager, position #0's fee growth is updated on Uniswap and 9 ERC-6909 tokens are minted with the remaining delta.
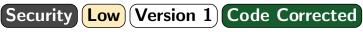
6. In the next transaction, some user deposits tokens to the vault. The fees are calculated again. Since position #1 has not been updated before, a manager fee of 1 token is calculated and another 9 ERC-6909 tokens are minted from the delta.

7. In total, 3 tokens in manager fees have been taken from 20 tokens of total fees.

Code corrected:

Fees are now only taken from positions that are getting changed (and therefore accounted in Uniswap) during the rebalance.

6.3 Allowance Abuse Through Hooks

Security Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-004

When adding liquidity, hooks are called that might modify the deposit deltas. As a consequence, malicious hooks could be used to drain user approvals.

Consider the following scenario:

1. A user wants to create a large deposit and sends a transaction. They added some margin to their approval to be sure that the transaction executes successfully.

2. A malicious executor sees it and sets a malicious pool with a post-add-liquidity hook and rebalances it so that there is one position.

3. The user's transaction arrives thereafter.

4. Liquidity is added to the position. The hook specifies an additional delta to receive. Optimally, the hook computes an amount so that the maximum amount pullable from the user will be taken by the module. Note that the hook can take the funds instantly.

5. The user has lost funds as only the regular delta is credited.

Note that in case of maximum approvals, the user's funds could be drained fully.

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

Code corrected:

The code has been adjusted so that no pools with post-add-liquidity hooks are allowed.

6.4 Bad Rounding

Design Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-005

`UniV4StandardModulePublic._deposit()` calculates the amount of liquidity necessary to be added to each position in the following way:

```
uint256 liquidity = FullMath.mulDiv(
    uint256(state.liquidity), proportion_, BASE
);
```

For token1, liquidity on Uniswap v4 is calculated with the following formula:

$L = \frac{\Delta Y}{\Delta \sqrt{P}}$

This indicates that liquidity values can be numbers significantly smaller than 1e18 if the following conditions apply:

- Token1 has low decimals.
- The `sqrtPrice` is high (either if token1 has a really low value or if token0 also has low decimals).
- The range of the position is large.

Consider the following example:

- Token0 has 6 decimals (e.g. USDC).
- Token1 has 6 decimals (e.g. USDT).
- 1 full token0 is worth exactly 1 full token1. The `sqrtPrice`, considering the given decimals, therefore is 1.
- The liquidity for 1000 token1 in a price range of 0.75 to 1.25 is: $\frac{1000e6}{\sqrt{1.25} - \sqrt{0.75}} \approx 4e9$

A user could mint a small proportion (~2.5e8) to the vault such that the liquidity calculated for this position is effectively 0. They would then gain free shares.

Note that `UnderlyingV4.getUnderlyingBalancesMint()` contains the same rounding error.

Code corrected:

`liquidity` is now rounded up.

6.5 Lack of Event Emissions

[Design] [Low] [Version 1] [Code Corrected]

CS-ARRKS-MOD-UNIV4-007

The following event emissions are missing:

- `LogDeposit` in `UniV4StandardModulePublic.deposit()`. Note that the event is commented out.
- `LogInitializePosition` in `UniV4StandardModulePosition()`. Note that the event is not defined but `ValantisHOTModule` emits such an event in the same function.

Code corrected:

- Corrected: `LogDeposit` is now emitted accordingly.
- `LogInitializePosition` seems no longer necessary as the associated state change is now always the same and therefore covered with the `LogSetModule` event in `setModule()`.

6.6 Manager Fee Finalization DoS

[Correctness] [Low] [Version 1] [Code Corrected]

CS-ARRKS-MOD-UNIV4-008

Updating the manager fee of a vault is subject to a waiting period. The process is thus divided into two different calls:

1. `ArrakisStandardManager.submitIncreaseManagerFeePIPS()`

2. `ArrakisStandardManager.finalizeIncreaseManagerFeePIPS()`

`finalizeIncreaseManagerFeePIPS()` calls `UniV4StandardModule.setManagerFeePIPS()` with the fee that has been passed to `submitIncreaseManagerFeePIPS()` before. `submitIncreaseManagerFeePIPS()`, however, does not perform a check that is done in `setManagerFeePIPS()`:

```
if (newFeePIPS_ > PIPS) revert NewFeesGtPIPS(newFeePIPS_);
```

If a fee greater than `PIPS` is submitted, it can never be finalized. Submissions can also not be altered, leading to a DoS of fee increases in the given vault.

Code corrected:

`ArrakisStandardManager.submitIncreaseManagerFeePIPS()` now checks that the new fee does not exceed `PIPS`.

6.7 Missing Init Minima

Design Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-010

`UniV4StandardModule.initialize()` sets the parameters `_init0` and `_init1` during initialization. These parameters are used to determine the ratio of the initially deposited tokens. If these values are set to 0, arbitrary amounts of vault shares can be minted for free. Note that this may DoS a vault (e.g. minting the maximum number of shares).

Code corrected:

The init values are now required to be non-zero.

6.8 Missing Range Check

Design Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-011

`ArrakisStandardManager.setModule()` allows executors to change the currently used module of a given vault. It calls the vault's `setModule()` function that changes the `module` storage variable to the address of the new module before calling `withdraw()` on the old module.

In the case that the new module is the `UniV4StandardModule` and the old module allows for some form of reentrancy (for example, this would also be possible in the `UniV4StandardModule` if it would allow hooks for removing liquidity), `ArrakisStandardManager.rebalance()` or `setPool()` could be called on the new module during a callback.

This would allow an executor to create new ranges (after donating some Uniswap ERC-6909 tokens to the module) before `initializePosition()` is called. Since `initializePosition()` does not check for any pre-existing ranges, the module can be changed successfully. On newly created vaults, this results in the `init0` / `init1` checks of the first deposit being bypassed.

Arrakis Finance - Uniswap V4 Module - ChainSecurity - © Decentralized Security AG   19

Code corrected:

While it is still possible to create positions before the initialization of a module, the first deposit check can no longer be bypassed this way as it is no longer dependent on the `_ranges` array.

6.9 No Accrual During Fee Change

`Correctness` `Low` `Version 1` `Code Corrected`

CS-ARRKS-MOD-UNIV4-012

`UniV4StandardModule.setManagerFeePIPS()` does not accrue manager fees before changing the fee. Thus, the new fee is applied to any past gains of a module's positions starting from the point of the last liquidity modification.

Code corrected:

The function has been adjusted to withdraw the manager balance before updating the fee.

6.10 Pausing Without Effects

`Correctness` `Low` `Version 1` `Code Corrected`

CS-ARRKS-MOD-UNIV4-013

The guardian may pause the module with `pause()`. However, no function besides `pause()` and `fund()` (in `UniV4StandardModulePrivate`) has the `whenNotPaused` modifier. Effectively, no functionality will be pausable.

Code corrected:

Now, `fund()`, `deposit()` and all manager functions are pausable. Note that is the expected behaviour according to Arrakis Finance.

6.11 Private Hook Unusable

`Correctness` `Low` `Version 1` `Code Corrected`

CS-ARRKS-MOD-UNIV4-014

`ArrakisPrivateHook` defines a hook that is called when liquidity is removed. `UniV4StandardModule`, which is supposed to be compatible with pools using the hook contract, does, however, not allow pools with a hook that is called when liquidity is removed.

Code corrected:

All hooks unsupported by the `UniV4StandardModule` are now unsupported by the `ArrakisPrivateHook`.

6.12 Uni V4 Resolver Ignores Token Order

Correctness **Low** **Version 1** **Code Corrected**

The return values of `UnderlyingV4.totalUnderlyingForMint()` are ordered according to the pool's currencies. However, `UniV4StandardModule.isInversed()` is not considered in the `UniV4StandardModuleResolver`. Thus, the resolver will confuse the tokens and currencies if the token order is inversed.

Code corrected:

Return values, max amounts and init values are now correctly swapped if `isInversed()` is `true`.

6.13 Uni V4 Resolver Math

Correctness **Low** **Version 1** **Code Corrected**

The `UniV4StandardModuleResolver` provides important values to the router. Namely, it specifies how many shares to mint and how many funds to pull from a user. However, the resolver could compute wrong results.

Problem 1: The first problem that occurs is that the rounding of the proportion computation differs from the rounding in the public vault. More specifically, on line #122, a number of shares to mint is computed and defined. Thereafter, the required token amounts are computed. The proportion is computed as follows:

```
uint256 proportion =
    FullMath.mulDiv(shareToMint, BASE, totalSupply);
```

In contrast, `ArrakisMetaVaultPublic.mint()` rounds up, favoring the system. Ultimately, in cases where the proportion is not perfectly rounded down, the returned token amounts might be too low for the `shareToMint` which could possibly result in the router approving less tokens than required.

Problem 2: The reduction of the deposit maxima by the number of ranges might not account for all rounding errors leading to a violation of the specified max amounts. The code below essentially ensures that `numberOfRanges` rounding errors in the final computation are tolerated:

```
uint256 numberOfRanges = _ranges.length;

if (
    numberOfRanges >= maxAmount0_
        || numberOfRanges >= maxAmount1_
) {
    revert MaxAmountsTooLow();
}

maxAmount0_ = maxAmount0_ - numberOfRanges;
maxAmount1_ = maxAmount1_ - numberOfRanges;
```

The code below specifies the amounts to pull from a user:

```
(amount0ToDeposit, amount1ToDeposit) = UnderlyingV4
    .totalUnderlyingForMint(underlyingPayload, proportion);
```

However, `totalUnderlyingForMint()` might have a higher error than `numberOfRanges`. The first reason being that the potential rounding up regarding the leftovers is not considered. Second, the errors of the `SqrtPriceMath` might have higher rounding errors than 1 (e.g. double division rounded up might have higher errors). Ultimately, the max amounts could be violated.

Code corrected:

The computation of the proportion is now done equivalently. The max amounts are now reduced sufficiently.

6.14 Wrong Counting of Minted and Burned Tokens

Correctness Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-019

`UniV4StandardModule._rebalance()` counts the minted and burned liquidity in both tokens. However, it does not consider the fees accrued when returning from `_addLiquidity()` and `_removeLiquidity()`. Namely, that is due to the `callerDelta` returning deltas including fees.

Consider the following scenario:

1. One position is defined with 10 tokens each at the current price. Assume that 2 tokens of fees have been earned in both tokens.

2. The position is rebalanced to remove half of the liquidity.

3. `_removeLiquidity()` will remove 5 of both tokens. The delta returned, however, will specify 7 tokens each due to fees being included.

4. The burned liquidity will result being 7 instead of 5 (as the fees are not burned liquidity from the position).

Note that if the fees had been collected before removing the liquidity, the result would have been 5 for both tokens. Thus, the accounting can be inconsistent.

Code corrected:

The code has been adjusted to not include the fees.

6.15 Wrong Liquidity Invariant

Correctness Low Version 1 Code Corrected

CS-ARRKS-MOD-UNIV4-020

`UniV4StandardModule._rebalance()` calls `_addLiquidity()` before making any delta adjustments regarding the fees. `_addLiquidity()` then performs the following checks right after the call to `PoolManager.modifyLiquidity()`:

```
if (currency0BalanceRaw_ > 0) revert InvalidCurrencyDelta();
uint256 currency0Balance =
    SafeCast.toUint256(-currency0BalanceRaw_);
if (currency1BalanceRaw_ > 0) revert InvalidCurrencyDelta();
uint256 currency1Balance =
    SafeCast.toUint256(-currency1BalanceRaw_);
```

However, since `modifyLiquidity()` adds the earned fees since the last position update to the delta, it can also be positive in case token amounts required for adding the given liquidity are smaller than the actual fees that have been accrued since the last update.

Removing liquidity beforehand can also exacerbate this problem as the delta becomes already positive before `_addLiquidity()` is called.

Code corrected:

Note that the checks in `_addLiquidity()` have been removed. Also note that the check was not strictly needed in the first place.

6.16 `UnderlyingV4` Ignores Management Fees for Mint Amounts

Correctness | Low | Version 1 | Code Corrected

CS-ARRKS-MOD-UNIV4-021

The minting-related functionality of `UnderlyingV4` (e.g. `totalUnderlyingForMint()`) ignores `self.managerFeePIPS()` so that the amounts needed for minting will be overestimated.

Code corrected:

The code has been adjusted to deduct the fees.

6.17 Incorrect Unused Leftovers

Informational | Version 2 | Code Corrected

CS-ARRKS-MOD-UNIV4-030

The leftovers computed in `UniV4StandardModulePublic._deposit()` on line 187 are incorrectly computed as they do not account for the `msg.value` that has been passed to the `UniV4StandardModulePublic.deposit()` function. However, the leftovers are unused as they do not impact the fee computations.

Code corrected:

Fees are now directly summed up from the `modifyLiquidity()` calls over all ranges. `UnderlyingV4.totalUnderlyingWithFees()` (and the respective leftovers) is no longer used in the `_deposit()` function.

6.18 Balance Function Errors

[Informational] [Version 1] [Code Corrected]

The `UniV4StandardModule` defines several `view` functions that return the current balances held. For example, `totalUnderlying()` returns the total amount of token0 and token1 held by the module. That accounts for LP fees and manager fees and returns the amounts in the order of token0 and token1 of the vault.

Other similar functions are inconsistent with that. Namely,

1. `totalUnderlyingAtPrice()` does not deduct the manager fees from the returned amounts,

2. `managerBalance0()` does not consider `isInversed` and might return the manager balance in token1,

3. and `managerBalance1()` does not consider `isInversed` and might return the manager balance in token0.

Code corrected:

All mentioned functions now correctly consider the `isInversed` flag.

6.19 Empty Return Values

[Informational] [Version 1] [Code Corrected]

`UniV4StandardModule.rebalance()` defines return values to which no values are assigned. However, the return values are also not parsed in `ArrakisStandardManager.rebalance()`.

Code corrected:

`UniV4StandardModule.rebalance()` now returns the return values of the internal function.

6.20 Irrelevant Actual Balances in Rebalance()

[Informational] [Version 1] [Code Corrected]

In the swap part of `UniV4Module._rebalance()`, the variables `balances.actual0` and `balances.actual1` are calculated after `take()` is done on the `PoolManager.take()`, however, will always send the exact specified amount or revert. Additionally, the values are then never used when setting the token allowances to the swap router.

Furthermore, the values are added to `expectedMinReturn` in the slippage check. This addition will always add 0 as the `actual` value of the token that has not been taken before is added:

```
if (swapPayload_.zeroForOne) {
    ...
    balances.actual0 = _token0.balanceOf(
        address(this)
    ) - balances.initBalance0;
```

```
        ...
    } else {
        ...
        balances.actual1 = _token1.balanceOf(
            address(this)
        ) - balances.initBalance1;
        ...
    }
    ...
    if (swapPayload_.zeroForOne) {
        if (
            balances.actual1
                + swapPayload_.expectedMinReturn
                > balances.balance1
        ) {
            revert SlippageTooHigh();
        }
    } else {
        if (
            balances.actual0
                + swapPayload_.expectedMinReturn
                > balances.balance0
        ) {
            revert SlippageTooHigh();
        }
    }
```

Code corrected:

The mentioned balances have been removed.

6.21 NatSpec Inaccuracies

Informational   Version 1   Code Corrected

CS-ARRKS-MOD-UNIV4-031

The NatSpec comments provide documentation about functions. Below is a list of NatSpec inaccuracies:

1. `UniV4StandardModulePublic.deposit()`: Specifies that `proportion_` is the number of shares that need to be added. However, it is the proportion of liquidity that needs to be added.

2. `UniV4StandardModule.initialize()`: Specifies that `init1_` is the initial amount provided to the Valantis module. However, it is the initial amount provided to the Uniswap V4 module.

Code corrected:

All mentioned items have been correctly adjusted.

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 Floating Pragma

`Informational` `Version 1` `Acknowledged`

Arrakis Finance uses a floating pragma solidity ^0.8.26. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. Further, the `foundry.toml` file does not specify a fixed compiler version either.

Note that for `UnderlyingV4`, a floating pragma may be suitable to ensure the reusability of the code in the future.

Acknowledged:

Arrakis Finance states:

> We need to be sure during deployment we are using the right solidity version 0.8.26.

7.2 Gas Inefficiencies

`Informational` `Version 1` `Code Partially Corrected`

The gas consumption can be reduced in several places. Below is a non-exhaustive list:

1. `_checkTicks()` is redundant. Namely, that is due to Uniswap v4 performing the same check in any `modifyLiquidity()` calls.

2. The module requests the full position info from the pool manager in `setPool()`, `_withdraw()`, `_removeLiquidity()` and `_deposit()`. However, only the liquidity is needed which could be requested through the library function `StateLibrary.getLiquidity()`. That would reduce the amount of storage reads. Further, in many occasions that would reduce the amount of memory writes to the `Position.State` struct, too.

3. `_deposit()` uses `UnderlyingV4.totalUnderlyingWithFees()` to compute the earned fees. However, the computation is not strictly necessary. Namely, the fees could be easily computed after all `modifyLiquidity()` calls by summing them up. Therefore, the storage reads and external call made for the `UnderlyingV4::totalUnderlyingWithFees()` function could be skipped if the action order in `_deposit()` were to be reordered. Note that this is similarly the case when removing all liquidity in `setPool()`.

4. Further, some computations of `UnderlyingV4.totalUnderlyingWithFees()` are not needed for functions `managerBalance0()` and `managerBalance1()`.

5. `sync()` on the pool manager is always called. However, it is not strictly needed for the native token, as `settle()` would be sufficient in such cases.

6. `UnderlyingV4.totalUnderlyingForMint()` queries `UniV4StandardModule.poolKey()` to retrieve the tokens from the key. However, given the assumption that the payload has the correct tokens, the token retrieval is redundant.

7. `totalUnderlyingWithFees()` calls `_totalUnderlyingWithFees()` without price which triggers the automatic price retrieval. However, the same price is retrieved in each iteration of the function.

8. The iteration in `_withdraw()` has complexity of $O(n^2)$. However, the complexity could be reduced to $O(n)$ if the iteration would start at the last item instead of the first one. Note that this has been reported in Version 2.

Code partially corrected:

The code has been partially corrected:

1. `_checkTicks()` has been removed completely.

2. The code now uses the library function `getPositionLiquidity()`.

3. `_deposit()` now calculates the fees from the return values fo `modifyLiquidity()`.

4. Not corrected.

5. `sync()` is now only called for non-native tokens.

6. Not corrected to potentially support modules with multiple pools in the future.

7. Not corrected to potentially support modules with multiple pools in the future.

8. Corrected: The computation has been improved significantly.

7.3 Inconsistent First Deposit Checks in Resolver

Informational Version 1 Code Partially Corrected Risk Accepted

CS-ARRKS-MOD-UNIV4-026

The resolver uses the total supply as a check whether the init values should be used (first deposit). However, that is inconsistent with the module which performs other checks. While typically the checks should be roughly equivalent, they can differ under certain circumstances. Thus, the resolver might return wrong results.

Please note that this change also requires a mitigation for Deposit frontrunning. Otherwise, the issue will also extend to deposits done via the router.

Code partially corrected:

The code has been generally corrected. Namely, the total supply check is typically valid now. However, the init values can be circumvented. This is possible by calling `setModule()` twice and donating tokens after the first call. Hence, in such cases, the supply will remain zero and the resolver could return wrong results.

Risk accepted:

Arrakis Finance states that the remaining issue will be mitigated by the team performing a small mint directly after deployment of a new vault, ensuring that the first deposit can not be performed by an executor.

7.4 Old Library Used

`UnderlyingV4.getUnderlyingBalances()` uses the library `LiquidityAmounts` to calculate the correct token amounts for Uniswap positions. The token amounts are meant for Uniswap v4 but the library is from a Uniswap v3 repository. In the event that some calculations change in the future, this will not be detected.

Similarly, `UnderlyingV4.getAmountsForDelta()` uses the library `SqrtPriceMath` from a Uniswap v3 repository to calculate the token deltas for a liquidity delta in a price range.

Code partially corrected:

`SqrtPriceMath` in `UnderlyingV4` has been updated to the v4 version. `LiquidityAmounts` is still a v3 library.

7.5 Uncommon Style Choices

The code deviates from the programming-style choices followed and from best practices on several occasions. Below is a non-exhaustive list:

1. `UniV4StandardModule.totalUnderlyingAtPrice()` could reuse `UniV4StandardModule._getPoolRanges()`. Generally, the usefulness of `_getPoolRanges()` is questionable, given that always the same pool is added to the structs. Thus, `UnderlyingV4` could simply receive the pool as an argument for a batch of positions on a given pool.

2. The `UnderlyingV4` library allows specifying multiple pools. However, the tokens might differ, resulting in the result being incorrect. Note that this point relates to the above.

3. The `UniV4StandardModule._unlockCallback()` function passes the `SUniswapV4.Withdraw` struct as an argument. However, four out of seven struct members are used as temporary variables in the function. Thus, it may be clearer to declare them in `UniV4StandardModule._withdraw()`.

4. Computing `UniV4StandardModule.isInversed()` from the vault's token0 and token1 on initialization could simplify the deployment process of the module for owners.

5. The `positionKey` used when getting the position information with `StateLibrary.getPositionInfo()` could be computed with `Positions.calculatePositionKey()` to reuse code.

6. `token0` and `token1` in `UniV4StandardModule._getTokens()` shadow the storage variables declared in `UniV4StandardModule`.

7. `UniV4StandardModule.validateRebalance()` handles the case where the pool's token1 is the native token. However, that is impossible. Further, for consistency with `UniV4StandardModule._initializePosition()`, `CurrencyLibrary.isAddressZero()` could be used.

Code partially corrected:

1. `_getPoolRanges()` is now reused (but still used).

2. No change.

3. No change.

4. No change.

5. Position keys are now computed using the appropriate library functions.

6. The variables have been renamed.

7. `validateRebalance()` now only checks if the first token is native.

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 2`

In `Version 2`, an approval feature has been added to the modules. Namely, the metavault owner can 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.

- The approved address should ensure that the control over the execution is only ever handed to trusted addresses. For example, if an approved address pulls ETH and performs a trusted swap where ETH is first sent out to an untrusted recipient, the recipient could mint shares at a discount. Namely, that is due to the leftover accounting not being accurate.

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

8.2 Hook Considerations

`Note` `Version 1`

The `ArrakisPrivateHook` implements a hook contract. Vault owners should be aware that flags are set as part of the address. The owners should ensure that the flags are set correctly. Otherwise, the hook will not provide the desired functionality.

Note that the Uniswap best practices suggest validating the hook flags directly in the constructor of the contract (e.g. Example Hook).

8.3 Module Oracle

`Note` `Version 1`

In `UniV4StandardModule.initialize()`, the oracle is not validated. Hence, it could be different from the oracle defined in `ArrakisStandardManager.vaultInfo`. Owners should ensure that the oracle is meaningful and optimally equal to the oracle defined in the standard manager.

Users and owners should be aware that an oracle returning zero would allow draining the module.

8.4 Native Token Constant

[Note] [Version 1]

`ArrakisStandardManager` contains immutable variables for the native token address and decimals. For no deployment should these immutable variables be different from the constants used in the `UniV4StandardModule`.

8.5 Out-of-Gas Possibility

[Note] [Version 1]

The `UniV4StandardModule` function keeps track of the tick ranges `_ranges`. Note that the size of that array is not restricted. Executors should be aware that increasing the size of the array might lead to potential out-of-gas scenarios where the module could be in a full DoS scenario. Note that this is due to every function relying on fully loading the array from storage.

8.6 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 only 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. While this prevents niftier attacks (like setting the operator of the module's balances in the `PoolManager`), executors can still use this call to siphon off tokens up to the maximum allowed slippage in every rebalance action. 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.

3. The executor performs a call to `PoolManager.donate()` with `amountIn = 1` and `expectedMinReturn = 1` (depending on the price) on a different pool (with the same tokens but their own custom hook that sends back 1 wei of token0/token1) and donates the maximum liquidity that still allows the slippage check in `ArrakisStandardManager` to pass. The donation is done to one of their own positions.

4. The executor performs a call to `PoolManager.modifyLiquidity()` with `amountIn = 1` and `expectedMinReturn = 1` (depending on the price) on a different pool (with one of the vault's tokens, another worthless token and their own custom hook that sends back 1 wei of token0/token1) and adds the maximum liquidity) that still allows the slippage check in `ArrakisStandardManager` to pass. One-sided liquidity for the vault token must be added and then traded against with the worthless token.

5. The executor performs a call to `PoolManager.modifyLiquidity()` with `amountIn = 1` and `expectedMinReturn = 1` (depending on the price) on a different pool (with the same tokens but their own custom hook that sends back 1 wei of token0/token1) and adds the maximum liquidity that still allows the slippage check in `ArrakisStandardManager` to pass. The pool must be highly imbalanced so that the executor can then trade against the position afterwards.

It is also worth noting that the first two points 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.

8.7 Supported Tokens

Note **Version 1**

The Uniswap V4 module supports a wide range of tokens which includes standard ERC-20 tokens and the native token. However, some token types are not supported. Below is a non-exhaustive list of such tokens:

1. Tokens with fees: Not supported due to the transfers to the pool manager providing an insufficient amount of funds to the fee taken.

2. Rebasing tokens: Not supported due to the lack of support for such tokens in Uniswap v4.

3. Reentrant tokens: Not supported as the core does not support such tokens (see core audit report).

4. ERC-1363: Tokens with a function such as `approveAndCall()` could allow an attacker to drain funds during a module migration. Assume that, during a migration, the `withdraw()` on a hypothetical module hands over the control over the execution flow to a malicious executor after some tokens have been sent to the Uniswap v4 module. Then, the attacking executor could rebalance the position (not reentrancy protected) to call `approveAndCall()` a token contract to hand out approvals and perform a "swap" correctly. Once rebalancing has successfully finished, the tokens could be pulled out of the module before the module is initialized with the position initialization. Ultimately, no slippage protection would catch such an attack.