PUBLIC

Code Assessment

of the Arrakis Modular Smart Contracts

July 5, 2024

Produced for

by

Contents

1 Executive Summary

Dear Spacing Guild team,

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

Spacing Guild implements an ecosystem of private and public vaults with strategies managed by the Arrakis backend. The vaults use so-called modules to integrate with a third-party system to implement the strategies. Currently, the only available module is an integration with Valantis HOT.

The most critical subjects covered in our audit are asset solvency, functional correctness and precision of arithmetic operations. Security regarding all the aforementioned subjects is good.

The general subjects covered are code complexity, gas efficiency, testing, and trustworthiness. Security regarding all the aformentioned subjects is satisfactory. However, the review brought to light the lack of thorough and meaningful testing, basic unit tests are done, but some of the bugs uncovered during the review could have been caught by proper end-to-end testing, see Rebasing Tokens Can Cripple the Functionality of Vaults and RouterSwapExecutor Cannot Swap to Native Token. We encourage Spacing Guild to implement a more complete test suite.

In summary, we find that the codebase provides a satisfactory 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	13
• Code Corrected	10
• Specification Changed	2
• Risk Accepted	1
Low-Severity Findings	15
• Code Corrected	8
• Code Partially Corrected	1
• Risk Accepted	2
• Acknowledged	4

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	08 Apr 2024	7aaba73a4f3f19f024baba4eb898c152d6e8d4f6	Initial Version
2	04 Jul 2024	98d2e2641aecd75e0dd6feb245e2294c083886b7	Version with fixes
3	05 Jul 2024	892c5588ee0f6544f6b35350796086b60b471bf2	Version with allowance fixes

For the solidity smart contracts, the compiler version `0.8.22` was chosen. After Version 2, the compiler version is `0.8.19`.

The following files are in the scope of the review:

```
ArrakisMetaVaultFactory.sol
ArrakisMetaVaultPrivate.sol
ArrakisMetaVaultPublic.sol
ArrakisPublicVaultRouter.sol
ArrakisStandardManager.sol
CreationCodePrivateVault.sol
CreationCodePublicVault.sol
Guardian.sol
ModulePrivateRegistry.sol
ModulePublicRegistry.sol
PALMVaultNFT.sol
RouterSwapExecutor.sol
TimeLock.sol
abstracts:
    ArrakisMetaVault.sol
    ModuleRegistry.sol
    ValantisSOTModule.sol
constants:
    CArrakis.sol
interfaces:
    AggregatorV3Interface.sol
    IArrakisLPModule.sol
    IArrakisLPModulePrivate.sol
    IArrakisLPModulePublic.sol
    IArrakisMetaVault.sol
    IArrakisMetaVaultFactory.sol
    IArrakisMetaVaultPrivate.sol
```

```
    IArrakisMetaVaultPublic.sol
    IArrakisPublicVaultRouter.sol
    IArrakisStandardManager.sol
    ICreationCode.sol
    IGuardian.sol
    IManager.sol
    IModulePrivateRegistry.sol
    IModulePublicRegistry.sol
    IModuleRegistry.sol
    IOracleWrapper.sol
    IOwnable.sol
    IPALMVaultNFT.sol
    IPermit2.sol
    IRouterSwapExecutor.sol
    ISOT.sol
    ISOTOracle.sol
    ISovereignPool.sol
    ITimeLock.sol
    IValantisSOTModule.sol
    IWETH9.sol
modules:
    SOTOracleWrapper.sol
    ValantisSOTModulePrivate.sol
    ValantisSOTModulePublic.sol
structs:
    SManager.sol
    SPermit2.sol
    SRouter.sol
```

After Version 2, the following changes have been made:

Deleted:

```
interfaces:
    ISOT.sol
    ISOTOracle.sol
```

Added:

```
structs:
    SValantis.sol
```

SOT has been renamed HOT.

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Furthermore, the external protocols and contracts with which the system integrates (Permit2, Valantis), as well as the third-party libraries employed in the source code (OpenZeppelin, Solady, Create3, UniswapV3 helpers) are out of the scope of this review and are expected to work always according to specification.

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.

Spacing Guild implements an on-chain ecosystem comprising an ensemble of asset vaults and the infrastructure needed to deploy them, integrate them with external protocols, and manage their funds. Vaults come in two distinct flavors - public and private - although both types share the characteristics of being essentially bound to one pair of tokens, and of being all managed by the same central fully-trusted entity (acting through a dedicated manager contract), which decides on the investment strategy to put in place using the provided assets, for a fee.

A public vault, as the name suggests, is open for anyone to contribute to, in exchange for *vault shares*, themselves tradable ERC-20 tokens, that can be redeemed against the appropriate fraction of the underlying assets.
The typical, and simplest example of a public vault is one that holds an LP position on some DEX, in a pool containing the vault's two tokens: the position is expanded/contracted as users proportionally deposit to/withdraw from the vault, and is rebalanced by the central manager in response to price movements, in order to keep it active and maximize the yield from trading fees. Said yield (minus the manager fee) becomes part of the redeemable assets, leading to the shares appreciating in value, according to the effectiveness of the manager's market-making strategy.

A private vault, on the other hand, is mainly conceived for new projects or token issuers who need to bootstrap the provision of their token to AMMs: the central manager frees them from the hassle of having to constantly rebalance for themselves the positions on the various DEXs as the price fluctuates.
For this use case, there is no need for open access to the vault, nor for shares representing one's contribution: the vault has an owner (the token issuer) who decides on a whitelist of addresses which are allowed to deposit and withdraw. Unlike for public vaults, deposits can be made at arbitrary ratios between the two tokens, independently of the ratio of the current vault's reserves.

In what follows, we expand on the architecture and the functionality of the on-chain system of smart contracts.

2.2.1 Modules

In order to integrate with external protocols, the system needs bespoke adaptors (known as modules in Arrakis parlance) exposing a standardized façade to the vaults for depositing and withdrawing funds.
Public modules - i.e. modules connecting public vaults with external protocols - expose a slightly different interface from private modules - used by private vaults: the function `IArrakisLPModulePublic.deposit()` just takes a `proportion` parameter, representing the desired "expansion factor" on the current reserves (therefore the token1/token0 ratio is preserved), whereas the function `IArrakisLpModulePrivate.fund()` accepts token transfers at any ratio. The rest of the interface is common to the two, including the `withdraw()` function (which, like `deposit()`, preserves the token ratio), and declared in `IArrakisLPModule`.

Modules also typically expose rebalancing functions to the central manager, but they are not standardized in the interface: some modules might have more than one, with varying semantics, some others might have none (e.g. an adaptor for UniswapV2); the manager is aware of the details of each module, and is able to call any of its functions with arbitrary granularity.
The functions to query and to withdraw the current outstanding manager fees, and to set the fee rate, are standardized. The same goes for the function `validateRebalance()`, used by the manager to check the current state (e.g. the spot price) against an oracle, before and after rebalancing.

Modules are deployed behind beacon proxies, to facilitate deployment and upgrades of multiple instances of the same logic with different parametrizations (e.g. token addresses). Each module instance is immutably linked to one and only one vault.
The beacons are critical components of the system, as they provide (and possibly change) the implementation addresses to the proxies; therefore, they need to be whitelisted for proxy use by a central admin of the system, who ensures their correctness and non-maliciousness (see Module registry for more).
Modules can be paused - except for the `withdraw()` function - by the central "guardian" of the system (see Guardian for more).

2.2.2 Valantis modules

At the current stage of the project, the only external integration of the system is with Valantis, a novel DEX design with a focus on modularity (notice that a security review of Valantis is out of scope for this audit). Of the many possible configurations that a Valantis pool can take, Arrakis chooses to only support one specific type, which we briefly outline below.

The `SovereignPool` is the contract actually holding the token reserves, and exposing the swap and flash-loan functionality. However, it is completely agnostic to the concrete swap logic chosen: implementing the mathematics of the reserves curve, by keeping track of state variables (like the current liquidity) and deciding on output amounts for swaps, is delegated to a second contract, the `ALM` (or liquidity module). The `ALM` is also the only address which can deposit/withdraw liquidity. The pool also recognizes a `poolManager` address, who is entitled to a cut of the swap fees; in the context of Arrakis' integration, the `poolManager` is the module instance.
The `SOT` (acronym for Solver Order Type) is a second contract, that acts as the `ALM` to the `SovereignPool`. The swap maths it implements is much akin to that of UniswapV3, but the LP logic has the notable peculiarity of only allowing a single position by a single `liquidityProvider`: in the context of Arrakis' integration, this address is the module instance, the same acting as `poolManager` to the pool. The position is only one, but can be rebalanced at will; deposits and withdrawals are allowed at any token ratio: the active liquidity is then set to the maximum attainable, and the excess reserves are considered as "passive" and do not take part in swaps.

The integration with Valantis is provided by two modules: `ValantisSOTModulePublic` and `ValantisSOTModulePrivate`, both of which inherit from the abstract `ValantisSOTModule`. The abstract contract implements most of the functions, while the two specialized ones implement the functions `deposit()` and `fund()`, respectively.
The main functions implemented in the abstract contract are:

- `withdraw()`: only callable by the Arrakis vault linked to the module. It withdraws the specified proportion of the two tokens from the pool (through the `SOT`), and then it forwards the tokens to the specified receiver.
- `withdrawManagerBalance()`: callable by anyone. Forwards the outstanding manager fees to the manager contract.
- `swap()`: only callable by the manager contract. This is one of the two rebalancing functions: it withdraws everything from the pool, then partially swaps one of the two tokens for the other, on a specified, arbitrary trading venue (e.g 1inch router), and finally re-deposits everything. This has the effect of leaving the position's price bounds and the pool's spot price unaltered, while adjusting the pool's active liquidity and passive reserves: it can be used, for example, if there is too much excess of one token, to swap part of that excess for an "equivalent" amount of the other token, so that the pool liquidity can increase.
- `setPriceBounds()`: only callable by the manager contract. This is the second of the two rebalancing functions: it simply sets a new price range for the Valantis position. This also causes the active liquidity and passive reserves to adjust according to the new bounds.

- `validateRebalance()`: called by the manager contract before and after rebalancing. Reverts if the pool's current spot price deviates by more than a specified threshold from the price given by the specified oracle.

The two specialized functions are:

- `ValantisSOTModulePublic.deposit()`: only callable by the vault linked to the module. Only allows depositing at the same token1/token0 ratio as the current pool reserves (active + passive). On first deposit, when both reserves are 0, an initial ratio of `init1/init0` is enforced, where `init0` and `init1` are defined at module initialization time.
- `ValantisSOTModulePrivate.fund()`: only callable by the vault linked to the module. Allows depositing at any ratio.

2.2.3 Vaults

Vaults are the users' entry points into the system, exposing an interface to deposit and withdraw, which maps onto the functions of the underlying module. Notably, vaults provide no rebalancing functions: the manager contract directly calls the module to perform its actions.
Public and private vaults are implemented in the `ArrakisMetaVaultPublic` and `ArrakisMetaVaultPrivate` contracts, respectively; their common functionality is implemented in the abstract `ArrakisMetaVault` contract, which we describe here.

At any given time, a vault is linked to one and only one module instance, called the *active module*. However, it also has a whitelist of other module instances that it *can* link to: it is up to the manager contract to call the `setModule()` function in order to switch from the current active module to another in the whitelist.
All vaults have an owner (see Public vaults and Private vaults for more details), that can add and remove modules from the whitelist. Adding a module actually deploys a new instance attached to the specified beacon (using the `createModule()` function in Module registry), permanently links it to the vault in question, and then adds it to the vault's whitelist. The vault's first active module is deployed and whitelisted at vault creation time (see Factory for more details).

2.2.4 Private vaults

The owner of a private vault is arbitrarily assigned at creation time by the deployer, and in typical scenarios will be an address belonging to the new project / token issuer. Ownership is transferrable, and tracked by a dedicated ERC-721 contract called `PALMVaultNFT`.
On top of the abstract vault logic, private vaults implement little more than a thin wrapper around the active module's `fund()` and `withdraw()` functions. Funding and withdrawing is restricted to a whitelist of *depositors*, that is decided upon by the owner.

2.2.5 Public vaults

The owner of a public vault is set at construction time to be a freshly-deployed Arrakis-controlled Timelock contract: ownership of the vault cannot be transferred or renounced.
The `mint()` and `burn()` functions implement a somewhat-standard tokenized vault behavior, as the vault itself is an ERC-20 contract. They take a desired number of shares as parameter, they compute the corresponding fraction of the total supply, and they use that as the `proportion` parameter for the active module's `deposit()` or `withdraw()` function, respectively. Thus, token deposits and withdrawals are always made at the current reserves ratio. Mitigations are also in place to thwart common inflation attacks.

2.2.6 Factory

The `ArrakisMetaVaultFactory` is the contract used to deploy public and private vaults. It is owned by the Arrakis admin. It is pausable by the owner, who can thus freeze deployment of new vaults. The owner also decides on a whitelist of *deployers*, who are the only ones who can deploy public vaults.
The contract keeps two sets with the public and the private vaults that it has deployed: these sets can be queried for inclusion using the convenience functions `isPublicVault()` and `isPrivateVault()`.
The deployment functions are:

- `deployPublicVault()`: only callable by a whitelisted deployer. It creates a new Timelock contract that will be the owner of the public vault: all the privileges (propose, cancel, execute, admin) on the Timelock are given to a deployer-provided address, which is assumed to be an Arrakis admin. Through the Module Registry, it also creates a new dedicated module instance out of the specified beacon. The vault is bound to the timelock and the module, the provided token addresses, as well as other context addresses (e.g. the manager and module registry), and is added to the set of public vaults. The manager contract is informed of this newly-created vault, through the `initManagement()` function (see Standard manager for more).
- `deployPrivateVaults()`: callable by anyone. Ownership of the vault is assigned to the provided arbitrary address, through the `PALMVaultNFT.mint()` function. As for the public vaults, a dedicated module instance is created, the vault is appropriately parametrized, and the manager contract is informed about the new vault.

2.2.7 Module registry

A module registry is in charge of the deployment of module instances (i.e. beacon proxies), and of keeping a whitelist of allowed beacons. It has an owner (an Arrakis admin), who decides on the whitelist of beacons.
There are two deployed registries: a `ModulePublicRegistry` and a `ModulePrivateRegistry`, although most of the functionality is common and implemented in the abstract `ModuleRegistry`. The only notable difference between the two is the concrete set of beacons they will end up whitelisting.
The main functions are:

- `createModule()`: deploys a proxy attached to the supplied beacon (provided it's whitelisted), and initializes it with the supplied opaque payload. Notice that one cannot anticipate the specifics of every future module that will be developed, therefore the initialization payload cannot but be an opaque byte array for the registry, and needs to be crafted appropriately by the vault deployer. However, as a "safety net" to catch *some* mistakes, this function checks, after initialization, that the module correctly recognizes the vault it is bound to, and the Guardian that can pause it.
- `whitelistBeacons()`: only callable by the owner, whitelists the supplied beacons. As was mentioned, beacons are critical components of the system, because they provide the address that will be delegatecalled by the module instances, therefore they need to be thoroughly evaluated by the system admins before being whitelisted. As a safety net, protecting against some *accidental* mistakes, this function checks that beacons recognize a pre-defined Arrakis address as their `owner()`.
- `blacklistBeacons()`: only callable by the owner, blacklists the supplied beacons.

2.2.8 Standard manager

This contract acts as a go-between for the Arrakis backend engine to interact with vaults and modules: it is the only address that is authorized to switch modules on a vault, to change a module's manager fee (fee increases are subject to a time-lock of one week through proposals that cannot be overridden during this time, this may results in an additional week delay if any error/typo is made), and to rebalance modules.
It is deployed behind a "regular" proxy, whose admin is an Arrakis address. It has an owner (also an Arrakis address), who can call it to set the manager fee on any vault's active module. It is pausable by the Guardian.

Its main functions are:

- `withdrawManagerBalance()`: calls the homonymous function on the active module of the specified vault (which transfers the manager fees to this contract), then forwards the tokens to the correct *receiver* (the owner can choose, through a dedicated `setReceiverByToken()` function, an appropriate receiver address for each token).
- `initManagement()`: only callable by the factory. Informs this contract about the creation of a new vault, so that it can start managing it. initializes the manager fee on the vault's active module.
- `updateVaultInfo`: only callable by the vault's owner. Sets important parameters that constrain the manager's room for maneuver, namely an *executor* address (the only one who can `rebalance()` that vault), an oracle, a maximum slippage, and a maximum deviation (used to validate rebalances), and a minimum cooldown period that has to elapse between subsequent rebalances.
- `rebalance()`: only callable by the *executor* previously chosen by the vault's owner. Performs a series of low-level calls into the vault's active module, with opaque byte-array payloads crafted by the Arrakis backend to properly encode the right function calls into the module-specific methods, conveying the desired rebalancing strategy. Before and after this batch of rebalancing operations, the module's `validateRebalance()` function is called (using as parameters the oracle and maximum deviation chosen by the vault's owner) as a measure against state manipulation of the integrated system. Additionally, the total value stored in the vault is gauged before and after the rebalance, and explicitly checked not to have varied by more than a threshold (the maximum slippage set by the vault's owner in `updateVaultInfo()`).

2.2.9 Guardian

This contract exposes the function `pauser()`, queried by many contracts in the system to know the address that (currently) can pause them. This address can be changed using the `setPauser()` function, reserved to the contract's owner (an Arrakis admin).

2.2.10 Public vault router

An untrusted helper contract, used to facilitate deposits and withdrawals for public vaults, and to add further safety checks to guard against slippage and/or manipulation.
The rationale is that the functions `mint()` and `burn()` exposed by public vaults, besides possibly being unintuitive for the end-user, offer no protection against manipulation of the underlying reserves, thus leaving the user exposed to the risk of depositing/withdrawing at skewed ratios. To solve these problems, the router exposes the function `addLiquidity()`, which enforces the specified minimum and maximum deposit amounts for the two tokens, as well as a minimum number of shares received, when minting. The function `removeLiquidity()` likewise enforces the desired minimum amounts withdrawn, when burning shares.
The contract also defines a wealth of variants of these two helpers - mainly `addLiquidity()` - to further ease the user experience. There are versions wrapping the received native token before minting (so as to support ETH from users), swapping part of the tokens provided (thus allowing "imbalanced" contributions from the user), and integrating with `Permit2` (removing the need to set a token allowance for the router).

2.2.11 Changes in V2

- The flow of `setModule()` was updated to have a mandatory call to the new `initializePosition()` function in the modules.

2.3 Trust model and assumptions

The administration of the system is mostly centralized: apart from private vaults, which are managed independently by their owners, all privileged operations are restricted to Arrakis-controlled addresses (although many are time-locked). Therefore, the system owner needs to be fully trusted by the users.

We assume the admins to parametrize the system correctly upon deployment.
Likewise, we assume the semantics of the EVM operations implemented by the targeted blockchains to be comparable to Ethereum Mainnet. Should there be significant discrepancies among them (e.g. uneven adoption of EIPs), the contracts' bytecode should be ensured to be uniformly supported across all targeted chains.
We consider every supported token to be trustworthy, meaning it cannot jeopardize the system's integrity. Tokens that implement transfer fees are typical examples of unsupported tokens.

The following roles can be identified in the system:

- users: fully untrusted.
- owner of the registries: trusted to whitelist beacons that are trusted. Expected to be Arrakis.
- beacon proxy owner: trusted to set and update the beacons to implementations that have been reviewed and are not malicious towards the users. Expected to be Arrakis.
- public vaults owners and executors: trusted to set the parameters and execute rebalancing in a non-adversarial way towards the users. Expected to be Arrakis.
- private vaults owners and executors: fully untrusted. We still expected the owner to trust the executor.
- guardian: trusted to pause/unpause the modules, factory, and standard manager in a non-adversarial way. Expected to be Arrakis.

Supported tokens: ERC-777, tokens with fees on transfer, and tokens with 0 decimals are not supported. We expect Spacing Guild to carefully review any token pairs that will be used with the oracle in accordance with Integer Representation of Price Has Low Precision.

New modules: we assume new modules will implement the following rules/invariants:

1. Cannot update `manager`
2. Cannot update `metaVault`
3. Modules must be able to handle funds depending on the Vault.setModule function.
4. Should be careful with exact balances checks for rebasing tokens
5. Only manager can update manager fee. Only function to change manager fee must be `setManagerFeePIPS`.
6. `totalUnderlying` cannot be manipulated, critical for read-only reentrancy. If the integrated system exposes unsafe functions, `totalUnderlying` should have a reentrancy lock.
7. Public modules should detect first deposit with supply==0 to align with the vault's definition of an empty strategy
8. Modules need to revert on withdrawing proportion == 0

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

- Registries Initialization Can Be Frontrun Risk Accepted

Low-Severity Findings	7

- Burn Does Not Revert on a Zero Proportion Risk Accepted
- Decreasing Manager Fee Can Lead to Status Quo Acknowledged
- Duplicated Code Code Partially Corrected
- Missing Input Sanitization Risk Accepted
- Switching From an Empty Module to a Valantis Module Will DOS the Public Vault Acknowledged
- Unnecessary Low-Level Call Acknowledged
- Unused Code Acknowledged

5.1 Registries Initialization Can Be Frontrun

Design Medium Version 1 Risk Accepted

CS-ARRAKISMOD-007

The module registries have an `initialize()` function to set the factory. The call to the function can be frontrun by an attacker to set an arbitrary address. This will force Spacing Guild to redeploy the system, incurring some gas costs.

Risk accepted:

Spacing Guild responded:

```
The deployment script will detect if someone has frontrun and then stop the deployment.
```

5.2 Burn Does Not Revert on a Zero Proportion

Design Low Version 1 Risk Accepted

CS-ARRAKISMOD-013

The `burn` method of `ArrakisMetaVaultPublic` does not revert when the `proportion` is zero. This results in a user burning shares without receiving any of the underlying tokens.

Risk accepted:

Spacing Guild responded:

```
Module is already checking if the proportion is 0.
```

The module that is currently implemented (`ValantisModulePublic`) implements a check and will revert if `proportion = 0`. Spacing Guild is expected to make sure new modules also follow that pattern. Spacing Guild said:

```
every future modules should be consistent by itself and have this check.
```

5.3 Decreasing Manager Fee Can Lead to Status Quo

Design | Low | Version 1 | Acknowledged

CS-ARRAKISMOD-016

The function `ArrakisStandardManager.decreaseManagerFeePIPS()` checks that the fee is strictly smaller than the current one in `PIPS`, but the actual applied fee is in `BIPS` for Valantis pools. It is then possible to call `ArrakisStandardManager.decreaseManagerFeePIPS()` with a value slightly smaller than the current fee, and the pool still having the same fee value.

Acknowledged:

Spacing Guild responded:

```
We are ok with it. We want to have PIPS precision on fees to be compatible with uniswap v4.
```

5.4 Duplicated Code

Design | Low | Version 1 | Code Partially Corrected

CS-ARRAKISMOD-017

Some functionalities are duplicated throughout the codebase. For gas cost, codebase maintenance and general understandability, it is good practice to avoid code duplication. Here is a non-exhaustive list of duplicated functionalities:

1. The `manager` check in `ValantisSOTModule.setManagerFeePIPS` is implemented by the `onlyManager()` modifier.
2. The functionality of `ArrakisPublicVaultRouter._permi2SwapAndAdd` and `ArrakisPublicVaultRouter._permit2Add` are similar and could be merged into one function.

3. The function `ArrakisMetaVaultFactory.getTokenSymbol()` can use the dedicated function `_append()` to build the symbol string.
4. The code blocks to send back tokens at the end of `ArrakisPublicVaultRouter.wrapAndSwapAndAddLiquidityPermit2()` and `ArrakisPublicVaultRouter.wrapAndSwapAndAddLiquidity()` are the same.
5. In `ValantisSOTModule.initialize()`, `TEN_PERCENT` can be used in place of `PIPS / 10`.
6. The `1 ether` in `ArrakisMetaVaultPublic.mint()` and `ArrakisPublicVaultRouter._getMintAmounts()` could be replaced by `BASE`.
7. Some of the checks and procedures done in `ArrakisMetaVaultRouter` can be extracted as modifiers or helper functions.

Code partially corrected:

1. `onlyManager()` modifier is now used.
2. Spacing Guild responded:

```
we are ok with the current implementation.
```

3. `_append()` is now used.
4. Spacing Guild responded:

```
we are ok with the current implementation.
```

5. `TEN_PERCENT` is now used.
6. `BASE` is now used
7. Spacing Guild responded:

```
we are ok with the current implementation.
```

5.5 Missing Input Sanitization

Design | Low | Version 1 | Risk Accepted

CS-ARRAKISMOD-021

Some functions in the codebase are missing proper input sanitization. Here is a non-exhaustive list:

1. The `defaultFeePIPS_` in the `constructor` of `ArrakisStandardManager`. A value that is too low/high could be problematic, and even block the vaults deployment if `defaultFeePIPS > PIPS`.
2. The function `ArrakisStandardManager._updateParamsChecks()` does not sanitize `maxDeviation`, `executor`, and `stratAnnouncer`. For example, if `maxDeviation` is set to a value that is too low, rebalancing is likely to fail.
3. In the function `ArrakisStandardManager._updateParamsChecks()`, if `maxSlippagePIPS` is set to `0`, rebalancing is likely to fail.
4. In the `initialize()` function of `ValantisModule`, if `maxSlippagePIPS` is set to `0`, `_checkMinReturn()` is likely to fail.

5. In `ValantisSOTModule.setALMAndManagerFees()`, additional checks could be made to ensure that the tokens in the module and the ALM match. If possible, this check should also be done for the oracle to avoid any misconfiguration.
6. In `ValantisSOTModule.setALMAndManagerFees()`, an additional check could be made to ensure the module is the ALM's liquidity provider.
7. The function `ArrakisPublicVaultRouter.getMintAmounts()` is missing the empty max amounts checks.

Risk accepted:

Spacing Guild responded:

> `We are ok with these specific missing sanitizations.`

5.6 Switching From an Empty Module to a Valantis Module Will DOS the Public Vault

Correctness **Low** **Version 1** **Acknowledged**

CS-ARRAKISMOD-014

The contract `ValantisHOTModulePublic` has a flag called `notFirstDeposit`, that starts out as false and is then set to true after the first deposit. This flag controls whether the pre-determined initial values `(init0, init1)` should be used in lieu of the current pool reserves, as a base to compute the proportional amounts to be paid by the depositor.
To correctly carry this information over, across module switches, the function `ValantisHOTModulePublic::initializePosition()` sets this flag to true. However, it does so unconditionally, irrespective of the state of the previous module; in particular, it does not check whether the previous module has ever been deposited to. Therefore, in cases where a public vault's module is immediately switched out of, before anyone could make a deposit (this can happen if the devs find the module to be buggy right after deployment), the new module will have the flag set to true, but the pool reserves will be empty.
If the pool's tokens are rebasing, this allows the first depositor to first make a small donation to the pool at an arbitrary ratio, and then deposit at that ratio: this is because Valantis pools do not cache reserves of rebasing tokens. In the case of non-rebasing tokens, the cached reserves will be used, which will always be 0: this will effectively make it impossible to deposit any money into the module, as the `deposit()` function will never pull any tokens.
The only remedies to such a situation would be to change the implementation in the module's beacon, switch to a new module (with a different logic), or re-deploy the public vault.

Acknowledged

Spacing Guild responded:

> `in this case we will just deploy a new freshly public vault.`

5.7 Unnecessary Low-Level Call

Design Low Version 1 Acknowledged

CS-ARRAKISMOD-024

The function `ArrakisMetaVaultPublic._deposit()` calls the `deposit()` selector of `IArrakisLPModulePublic`, but in this case there is no need to add this complexity, as the selector is fixed and can be called directly.

The same is true for `ArrakisMetaVaultPrivate._fund()`.

Acknowledged:

Spacing Guild acknowledged and answered:

```
We want to use functionCallWithValue of Address.
```

5.8 Unused Code

Design Low Version 1 Acknowledged

CS-ARRAKISMOD-025

For codebase maintainability and comprehension, unused code should be removed. The constant `CArrakis.NATIVE_COIN`, for example, is never used.

Acknowledged:

Spacing Guild replied:

```
We decide to keep it, because it's used on the test folder and it will be used on future modules
```

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	12

- Dust in Allowance Can DOS the System Code Corrected
- Deposited Amounts Can Be Less Than proportion Code Corrected
- Initial Ratio Can Be Bypassed for Rebasing Tokens Code Corrected
- Integer Representation of Price Has Low Precision Specification Changed
- Minted Shares and Tokens Amounts Discrepancies Code Corrected
- Read-only Reentrancy Specification Changed
- Rebasing Tokens Can Cripple the Functionality of Vaults Code Corrected
- Shares Are Too Cheap Code Corrected
- Solidity Version Is Not Multichain Compatible Code Corrected
- Wrong Token Flow on setModule Code Corrected
- ISOTOracle Interface Is Wrong Code Corrected
- RouterSwapExecutor Cannot Swap to Native Token Code Corrected

Low-Severity Findings	8

- Cannot Reset to Default Receiver Code Corrected
- Inconsistent Definition of Empty Strategy Code Corrected
- Inherited Contracts Not Initialized Code Corrected
- Initializer Not Disabled in the Implementations Code Corrected
- Remaining TODOs Code Corrected
- Timelock Minimum Delay May Be Too Short Code Corrected
- Using Floating Pragma Solidity ^0.8.20 Code Corrected
- PALMVaultNFT Doesn'T Use the _safeMint Function to Mint Code Corrected

Informational Findings	1

- Wrong Natspec Code Corrected

6.1 Dust in Allowance Can DOS the System

Design Medium Version 2 Code Corrected

CS-ARRAKISMOD-033

In Version 2, the version of OpenZeppelin libraries was downgraded from v5.0.1 to v4.9.5. In the event of a rebasing token having the same behavior as USDT on approval (i.e., the allowance must be set to 0 before setting any new amount), the router, the vaults and the modules could stop working for that token because of potential dust in the allowance. The version 4.9.5 of OpenZeppelin's `SafeERC20` for `safeIncreaseAllowance()` does not reset the allowance to 0 before setting a new amount.

Code corrected:

The calls to `safeIncreaseAllowance()` have been replace by calls to `forceApprove()`.

6.2 Deposited Amounts Can Be Less Than `proportion`

Design | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-001

In the function `ValantisModulePublic.deposit()`, the `amount0/1` are rounded down and could represent a ratio that is smaller than `proportion` when compared to `_amt0/1`. This is bad for the protocol as it is making the vault shares cheaper than they should be, since `proportion` represents the ratio of new shares, when used during a `ArrakisMetaVaultPublic.mint()` call. This issue is made worse as the token decimals are low.

Code corrected:

`amount0/1` are now rounded up using `mulDivRoundingUp`.

6.3 Initial Ratio Can Be Bypassed for Rebasing Tokens

Design | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-002

The initial ratio `init0/init1` can be bypassed if at least one of the tokens in the `SovereignPool` is rebasing. This is due to the pool relying on its balance for rebasing tokens, instead of its internal accounting. An attacker could front run the first deposit in the pool by sending a small amount of rebasing token, skewing the first deposit. To recover from this, the vault manager has to rebalance the liquidity.

Code corrected:

A flag has been added to indicate whether a deposit will be the first. If the flag is true, the module will first remove any liquidity from the pool and send it to the `ArrakisStandardManager`, and then proceed to add the tokens in the `init0/init1` ratio.

6.4 Integer Representation of Price Has Low Precision

Design | Medium | Version 1 | Specification Changed

CS-ARRAKISMOD-003

The function `getPrice0()` calculates `price0`, which is essentially a quote for 1 whole token0 (`10**decimals0` wei) expressed in weis of token1; it does so starting from `priceX96`, a standard Q64.96 representation of the square-root price of token0/token1, coming from an oracle. However, while `priceX96` has enough precision to adequately represent any value observable in practice, `price0` does not, because the quote is rounded down to an integer. This incurs severe rounding errors whenever the price is very low; in the worst cases, it is rounded down to 0. Consider for example token0 to be SHIB (value of $0.00002343 as of this writing) and token1 to be GUSD (a stablecoin with only two decimals): clearly, an amount of one token0 is worth less than one wei of token1 (i.e. one cent), therefore the output of `getPrice0()` will be 0.
This makes `price0` unsuited for use as an equivalent representation of the token0/token1 price. However, the function `ArrakisStandardManager.rebalance()` uses it to calculate a quote on `amount0`, the amount of token0 held by the vault, in terms of token1; it does this to gauge the total value held in the pool, and ensure that it does not decrease too much as a result of rebalancing. Moreover, the function `ValantisSOTModule.validateRebalance()` applies the same quoting logic to the current spot price, and compares the result with the output of `getPrice0()`, to ensure that the spot price is not manipulated. Both these crucial checks are therefore broken; in the aforementioned SHIB/GUSD case, these functions will almost never detect any abnormal deviation, as they will be comparing 0 with 0.

Specification changed:

The project will not support token pairs exhibiting such poor price-rounding behaviour.

Spacing Guild responded:

```
We will not support token pair having this type of precision.
```

6.5 Minted Shares and Tokens Amounts Discrepancies

Correctness | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-004

The way the minted shares and corresponding token amounts are calculated are different in the `ArrakisPublicVaultRouter`, `ArrakisMetaVaultPublic`, and `ValantisSOTModule`. This discrepancy has multiple causes and effects listed below, ranging from not giving enough shares to the user to blocking the router.

1. When minting the first shares, the `ArrakisMetaVaultPublic` removes `MINIMUM_LIQUIDITY` from the shares distributed to the first LP. In comparison, when `ArrakisPublicVaultRouter` computes `_getMintAmounts`, the `MINIMUM_LIQUIDITY` is not removed from the shares for the first LP. This means that for the first user of a vault, the slippage protection implemented by the `ArrakisPublicVaultRouter` may succeed, but the final user could get less shares than the specified `amountSharesMin`.
2. The precision used in the contracts differs, one is `PIPS`, the others are `BASE`. This has the same impact as the issue described below.

3. The way the `proportion` is computed differs across the contracts. In the router, it is based on the token amounts, in the vault it is based on the shares. This means that going from `amount->proportion->shares->proportion'->amount'` can yield `proportion != proportion'` and thus `amount != amount'`. This has the effect of blocking all the helper functions for providing liquidity in the router under certain conditions, as the `ValantisSOTModulePublic` may pull more or less than expected, making the transaction revert.

Code corrected:

1. Spacing Guild responded:

   ```
   it's the wanted behaviour and we are ok with the current implementation.
   ```

2. The precision has been updated in `ArrakisPublicVaultRouter` to be BASE. BASE is now used everywhere some shares or proportions are computed.
3. The steps from the `ArrakisMetaVaultPublic` have been replicated in `ArrakisPublicVaultRouter`. Except for the first deposit, the values `_getMintAmounts()` outputs are now the same as in `ArrakisMetaVaultPublic.mint()` and `ValantisSOTModulePublic.deposit()`.

6.6 Read-only Reentrancy

Design Medium Version 1 Specification Changed

CS-ARRAKISMOD-005

In the case of a token is reentrant (e.g., ERC777), depending on the system integrated by the module, there is a possible read-only reentrancy in the `totalUnderlying` function, as shares are burned already some of the tokens are still in the pool.

The issue arises when the integrated system sends the tokens before fully updating its state. As the shares have been burned already, the value of a vault's share can be over-evaluated.

As in Version 1, the only implemented integration is Valantis SOT, this holds only if a pair contains at least one token that is reentrant and rebasing.

Specification changed:

Reentrant tokens will not be supported in the project.

Spacing Guild responded:

```
Will not support ERC777.
```

6.7 Rebasing Tokens Can Cripple the Functionality of Vaults

Design Medium Version 1 Code Corrected

CS-ARRAKISMOD-006

Rebasing tokens, which are meant to be supported by Arrakis Modular, break common assumptions about the behavior of the functions they expose: the root cause is that the amount of tokens transferred does not necessarily equate the resulting balance difference, at either the sending or the receiving end. Some parts of the source code employ such wrong assumptions to perform important calculations and safety checks: as a result, much of the system's functionality is gravely impaired.
Below is a list of functions affected:

1. `ArrakisMetaVault.setModule()`: the assertion that the old module is completely empty might fail. This is because, for many rebasing tokens, transferring out one's entire balance (as happens when withdrawing from the old module) may still leave some "dust" behind.
2. `ValantisSOTModule.swap()`: the final assertions are not guaranteed to succeed. The reason is a generalization of the previous point: transferring out `balanceOf(this) - initBalance` (which is how the amounts to deposit into the ALM are computed) is not guaranteed to bring the balance back to `initBalance`.
3. `ArrakisPublicVaultRouter`: the functions `_swapAndAddLiquiditySendBackLeftOver()`, `wrapAndSwapAndAddLiquidity()`, and `wrapAndSwapAndAddLiquidityPermit2` all include (duplicated) logic for sending back leftover tokens. The formulas computing the amounts to send back, however, may cause the `safeTransfer()` to revert for insufficient balance.
4. `ArrakisPublicVaultRouter._addLiquidity()`: the final assertions may fail, as the transferred amount is not guaranteed to be equal to the balance difference.

For completeness, note that several functions in the code (e.g., `mint()` and `burn()` in `ArrakisMetaVaultPublic`) emit events to log the deposited / withdrawn amounts: it is worth mentioning that these amounts do not exactly reflect the resulting differences in token balances.

Code corrected:

1. The assertion has been removed.
2. The assertions block has been removed.
3. Any leftover tokens (contract's balance) are now sent back to the user.
4. The assertion has been removed.

6.8 Shares Are Too Cheap

Design | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-008

When computing the `proportion` in `ArrakisMetaVaultPublic.mint()`, the value that represents what the vault will get (`proportion`) is rounded down, while the true proportion of the minted shares is not. Because of rounding errors, it may happen that the proportion is off by 1 wei, i.e. $\frac{shares}{totalSupply} \geq proportion$. This is unsafe for the protocol since it will get less value than it should.

Code corrected:

`ArrakisMetaVaultPublic.mint()` is now using `mulDivRoundingUp` to compute the proportion.

6.9 Solidity Version Is Not Multichain Compatible

Design | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-009

The chosen Solidity version, `0.8.22`, already implements the `PUSH0` opcode. This opcode is not supported on all the chains yet and could prevent deployment, or prevent use of the system after deployment.

Code corrected:

The Solidity version has been downgraded to `0.8.19`.

6.10 Wrong Token Flow on `setModule`

Correctness | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-010

In the function `ArrakisMetaVault.setModule()`, the old module sends the tokens to the new module via the `withdraw()` function, but the new module does not necessarily expect to hold tokens. With the current implementation of the `Valantis` integration there could be one of the following cases (non-exhaustive):

1. `setModule` is called with an empty payload: the funds will be stuck in the module until the beacon admin updates the implementation to unlock the funds.
2. `setModule` is called with a payload that does not consider the transferred funds: the funds will be stuck in the module until the beacon admin updates the implementation to unlock the funds.
3. `setModule` is called with a payload to deposit the transferred funds in the new module: the call will revert as the module cannot transfer the tokens from the caller.

Code corrected:

A new function `IArrakisLPModule.initializePosition()` has been added, in the current implementation (`ValatisHOTModule`), it deposits the balances of `token0/token1` in the Valantis ALM. The flow for `setModule()` has been updated as follows: the old module still withdraws everything and sends all the tokens to the new module, but now the flow forces a call to `initializePosition()`.

6.11 `ISOTOracle` Interface Is Wrong

Correctness | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-011

The interface `ISOTOracle` exposes functions that are not implemented in the `SOTOracle` contract. This could lead to calls reverting as the called function does not exist, breaking a functionality of the system, or the system itself. In the current implementation of Arrakis Modular, only functions that are implemented in the `SOTOracle` are called. To avoid this, it is good practice to import or reuse the interfaces provided by the integrated systems.

Code corrected:

This interface isn't used anymore and was removed.

6.12 `RouterSwapExecutor` Cannot Swap to Native Token

Correctness | Medium | Version 1 | Code Corrected

CS-ARRAKISMOD-012

The contract `RouterSwapExecutor`, which is a generic swap executor, cannot receive native tokens from a swap due to the lack of `receive` or `fallback` function. This will cause swaps to the native token to fail.

Code corrected: A `receive()` function has been added to the contract.

6.13 Cannot Reset to Default Receiver

Design | Low | Version 1 | Code Corrected

CS-ARRAKISMOD-015

In `ArrakisStandardManager.setReceiverByToken`, once a token receiver is set, the default receiver cannot be used anymore for that token. To be precise, one can set the token receiver to be the *current* default one, but there is no way to unset it and have it then always track the default receiver as it changes. This is because the `receiver_` parameter of this function cannot be zero.

Code corrected:

The function `ArrakisStandardManager.setReceiverByToken()` has been updated and now allows to set the receiver to `address(0)` to reset it to the default receiver.

6.14 Inconsistent Definition of Empty Strategy

Correctness | Low | Version 1 | Code Corrected

CS-ARRAKISMOD-018

The `ArrakisPublicVaultRouter` / `ValantisSOTModulePublic` and `ArrakisMetaVault` have different ways to detect whether a deposit is the first. The first one checks if both underlying tokens' balances are zero, and the second one relies on the total supply. In the case of the `Valantis` integration, the two solutions are not equivalent as they will not yield the same result in the case of rebasing tokens, as one can manipulate the rebasing tokens balances by sending some to the pool.

A secondary effect of this issue is the temporary DOS of the router for that vault for the first deposit, as the `supply` will be zero but `amount0/1` will not. This can be resolved by depositing through the vault directly.

Code corrected:

The check for the empty strategy is now based on the `supply` in `ArrakisPublicVaultRouter` and `ArrakisMetaVault`. A flag has been added in `ValantisSOTModulePublic` to indicate the empty strategy.

6.15 Inherited Contracts Not Initialized

Design Low Version 1 Code Corrected

CS-ARRAKISMOD-019

The contract `ValantisSOTModule` extends `PausableUpgradeable` and `ReentrancyGuardUpgradeable`, which are `Initializable`. But the function `ValantisSOTModule.initialize()` is not initializing the two inherited contracts.

Code corrected:

The constructor has been updated to call `__Pausable_init()` and `__ReentrancyGuard_init()`.

6.16 Initializer Not Disabled in the Implementations

Design Low Version 1 Code Corrected

CS-ARRAKISMOD-020

The proxied contracts `ValantisModule` and `ArrakisStandardManager` allow their implementations to be initialized after deployment. Even though we could not find any issue related to initializing the implementation of those contracts, it is usually a good practice to disable the initializers on the implementation.

Code corrected:

A call to `_disableInitializers()` has been added in both `ValantisModule` and `ArrakisStandardManager` to disable the initializers within the constructor.

6.17 Remaining TODOs

Design Low Version 1 Code Corrected

CS-ARRAKISMOD-022

There are remaining TODOs in the codebase that should be resolved before deployment.

Code corrected:

TODOs have been removed from the codebase.

6.18 Timelock Minimum Delay May Be Too Short

Design Low Version 1 Code Corrected

CS-ARRAKISMOD-023

The currently set minimum timelock delay for the public vaults owners is 1 minute. This may be a too short time frame for users to react to the change.

Code corrected:

The minimum delay has been updated to 2 days.

6.19 Using Floating Pragma Solidity ^0.8.20

Security Low Version 1 Code Corrected

CS-ARRAKISMOD-026

Contracts should be deployed with the same compiler version that has 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.

Code corrected:

`0.8.19` is used now.

6.20 `PALMVaultNFT` Doesn'T Use the `_safeMint` Function to Mint

Design Low Version 1 Code Corrected

CS-ARRAKISMOD-027

Usage of the `_safeMint` function is recommended over the `_mint` function to prevent the minting of NFTs to contracts that are not compatible with the ERC721 standard. The `_safeMint` function checks if the recipient of the NFT is a contract and if it supports the ERC721Receiver interface. If the recipient is a contract and does not support the ERC721Receiver interface, the minting operation will fail.

The OZ library specifies:

WARNING: Usage of this method is discouraged, use {_safeMint} whenever possible

---

Code corrected:

`_safeMint` is now used instead of `_mint` in the `mint` function of the `PALMVaultNFT` contract.

6.21 Wrong Natspec

Informational Version 1 Code Corrected

CS-ARRAKISMOD-031

The following natspec are incorrect:

1. In `ValantisModule.swap` and `IArrakisLPModule.swap`, the `@param router_` is described to be the address of the `RouterSwapExecutor`, but it should be arbitrary routers except the `RouterSwapExecutor`.
2. In `ArrakisMetaVaultFactory.blacklistDeployer` and `IArrakisMetaVaultFactory.blacklistDeployer`, the `@param deployers_` is described to be the list of addresses that the owner wants to grant permission to deploy. However, in this case, we want to revoke permission to deploy for the given addresses.

3. In `ValantisModule.withdraw` and `IArrakisLPModule.withdraw`, the `@param proportion_` is described to be the number of shares needed to be withdrawn, but it should be the proportion of the position.
4. `interest` mistyped `iPnterest` in the natspec of `ArrakisPublicVaultRouter.addLiquidity`, `ArrakisPublicVaultRouter.wrapAndAddLiquidity`, `IArrakisPublicVaultRouter.addLiquidity` and `IArrakisPublicVaultRouter.wrapAndAddLiquidity`.

Code corrected:

All the points above have been addressed.

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 Gas Optimizations

Informational **Version 1** Acknowledged

CS-ARRAKISMOD-028

The following gas inefficiencies can be improved:

1. Redundant modifier `whenNotPaused` in `pause()` and `whenPaused` in `unpause()` functions. This modifier is already used in the underlying `Pausable` or `PausableUpgradeable` contract, so it is redundant and can be removed in the following contracts: `ValantisSOTModule`, `ArrakisMetaVaultFactory`, `ArrakisPublicVaultRouter`, and `ArrakisStandardManager`.
2. The pattern `EnumerableSet.contains()` followed by `EnumerableSet.add()/remove()` can be optimized as `add/remove` returns whether the element was part of the set already or not.
3. The structs `VaultInfo` and `FeeIncrease` can use smaller data types for the time-related fields and leverage storage packing.
4. The storage variable `admin` in `ModuleRegistry` can be `immutable`.
5. The storage variable `manager` in `ArrakisMetaVault` can be `immutable`.
6. In the functions `ArrakisPublicVaultRouter.wrapAndAddLiquidity()` and `ArrakisPublicVaultRouter.wrapAndSwapAndAddLiquidity()`, only one of the tokens will not be `WETH`, so the logic where the second token is transferred to the router can be an `if-else` block instead of `if-if`.
7. Some computations with the pattern `if(a > b) { ... (a-b) ... (a-b)}` can cache the result of `(a-b)` and execute the operation in an `unchecked` block to save gas. Example: `ArrakisPublicVaultRouter.wrapAndSwapAndAddLiquidity()`, `ArrakisPublicVaultRouter.wrapAndAddLiquidityPermit2()`
8. In the functions `ArrakisPublicVaultRouter._permit2Add()` and `ArrakisPublicVaultRouter._permit2SwapAndAdd()`, the `tokenPermission.token` can only be either `token0` or `token1` so the `if-if` block in the `for` loop can be `if-elif-else`.
9. The function `initializeTokens` in `ArrakisMetaVault` could be merged with the constructor, allowing `token0` and `token1` to be immutable and removing the initialization test (`token0 != address(0) || token1 != address(0)`).
10. Parameters sanitization checks done in `initializeTokens` (in `ArrakisMetaVault`) could be reduced from 4 to 2.
11. `ArrakisMetaVaultPublic.burn` has no need to check if `shares_ > supply` since it will revert later if one tries to burn more than his balance. Avoid repetitive checks for others.
12. The check `_checkVaultNotAddressZero(vault_)` is redundant with `factory.isPrivateVault(vault_)` in `ModulePrivateRegistry`. This is also applicable to `ModulePublicRegistry`.
13. In `ArrakisPublicVaultRouter._swapAndAddLiquiditySendBackLeftOver`, since the router is not expected to hold any balance between two transactions, the contract balance could be returned instead of computing the exact amount.

14. For proxies, the CBOR metadata represents a good part of the deployed bytecode. The CBOR metadata, or only the metadata hash, can be disabled to reduce the gas cost on deployment.
15. Some checks can be moved earlier in the functions to revert early and save gas. Examples: zero address check in `ArrakisMetaVaultPublic.mint` and `ArrakisMetaVaultPublic.burn`
16. In `ArrakisPublicVaultRouter` there are early wrappings that should be deferred until after the checks. Additionally, when adding liquidity without a swap, it is more efficient to wrap only the exact computed amount to avoid the need to unwrap any surplus.
17. The functions `getUnderlying()` and `totalUnderlyingAtPrice()` from `ValantisSOTModule` initialize the values `amount0` and `amount0` but never use them

---

Acknowledged:

Spacing Guild corrected some of the inefficiencies and acknowledged the rest.

1. The redundant modifier was removed.
2. Not implemented.
3. Not implemented.
4. `admin` is now `immutable`.
5. `manager` is now `immutable`.
6. Not implemented.
7. An unchecked block was added.
8. Not implemented.
9. `initializeTokens` was merged.
10. Not implemented.
11. The check was removed.
12. Not implemented.
13. Returns the contract balance.
14. Not implemented.
15. Not implemented.
16. Not implemented.
17. Corrected.

7.2 Inconsistent Error Naming

Informational **Version 1** **Acknowledged**

CS-ARRAKISMOD-029

Inconsistent error naming in `ArrakisPublicVaultRouter.swapAndAddLiquidity`. The `msg.value` is checked to be different from the `amountMax` and if it is, the execution context reverts with `NotEnoughNativeTokenSent`. However, if the `msg.value` is greater than the `amountMax`, not enough native tokens have been sent, but the context still reverts with the same error.

Acknowledged:

Spacing Guild acknowledged and is aware of the issue.

7.3 Multi-chain Compatibility

Informational **Version 1** **Acknowledged**

CS-ARRAKISMOD-030

The project aims to be a multi-chain compatible platform, but there are some inconsistencies in `ArrakisPublicVaultRouter`:

1. The `weth` address, which represents the wrapped version of the native token of the chain, is expected to implement the `IWETH9` interface, which may not be the case on all chains. Spacing Guild must carefully choose the chains Arrakis Modular will be deployed to, as any integration with the native token may break if its wrapped version does not implement `IWETH9`.
2. Some functions can revert with `NoWethToken` which is not really appropriate for multi-chain.

7.4 `solady` Is Experimental

Informational **Version 1** **Risk Accepted**

CS-ARRAKISMOD-032

The codebase uses the `solady` contracts and library extensively. It is important to note that the code is still experimental, as the README file in the project highlights:

```
This is **experimental software** and is provided on an "as is" and "as available" basis.

We **do not give any warranties** and **will not be liable for any loss** incurred through any use of this codebase.
```

Risk accepted:

Spacing Guild accepted the risk and is aware of this potential issue.

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 Users Should Use the Router for Slippage Protection

Note Version 1

`ArrakisPublicVaultRouter` does not only provide an easy way for the user to interact with Arrakis with a higher level of abstraction but also provides a slippage protection mechanism. This mechanism is implemented through hard limit checks on the minimal (and maximal) amount of tokens that the user will receive.

This also limits the effect of sandwich attacks, as the router will revert the transaction if the slippage is too high. This also applies to the amount of minted shares.