ENIGMA DARK

Securing the Shadows

Security Review

Arrakis

Public Uniswap v4 Module

July, 2025

Contents

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Arrakis Modular: Uniswap V4 Public Module
Arrakis Modular is a DEX-agnostic liquidity management framework built around Meta-Vaults and standardized modules. It enables integration with any two-token DEX (like Uniswap V4, Balancer, Ambient) through reusable, and upgradeable components. Designed for flexibility and scalability, it supports both public and private vaults, simplifies liquidity provisioning, and lays the groundwork for advanced strategies across DeFi.

Engagement Overview

Over the course of 1.6 weeks (1 week and 3 days), beginning July 4 2025, the Enigma Dark team conducted a security review of the Arrakis Modular: Uniswap V4 Public Module project. The review was performed by two Lead Security Researchers: 0xWeiss and vnmrtz.

The following repositories were reviewed at the specified commits:

Repository	Commit
ArrakisFinance/arrakis-modular	`f8a3cbd156325a54f5883edcb468126e4c55640a`

The scope of the review covered the following files:

```
src
├── modules
|   ├── UniV4StandardModulePublic
|   └── resolvers
|       └── UniV4StandardModuleResolver
└── ArrakisPublicVaultRouterV2
```

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	5	0	5
Informational	3	0	3

Findings

Index	Issue Title	Status
L-01	`ArrakisPublicVaultRouterV2` `addLiquidity` Flow Does Not Enforce Slippage Checks Post-Execution	Acknowledged
L-02	Minimum amount check on resolver makes one-sided liquidity not possible	Acknowledged
L-03	Missing event emission when updating the swap executor	Acknowledged
L-04	Integrators can be affected by lack of vault validation	Acknowledged
L-05	`amountInSwap` can be bigger than `amountMax`	Acknowledged
I-01	Minor improvements to code and comments	Acknowledged
I-02	Redundant deposit check	Acknowledged
I-03	Consider re-structuring the pausing roles	Acknowledged

Detailed Findings

High Risk

No issues found.

Medium Risk

No issues found.

Low Risk

L-01 - `ArrakisPublicVaultRouterV2` `addLiquidity` Flow Does Not Enforce Slippage Checks Post-Execution

Severity: Low Risk

Context:

- ArrakisPublicVaultRouterV2.sol#L201-L209
- ArrakisPublicVaultRouterV2.sol#L1075-L1087

Technical Details:
According to the Uniswap v4 best practices for custom routers, slippage protection must be enforced after any liquidity modification, using the actual delta values returned by the `modifyLiquidity` call.

`ArrakisPublicVaultRouterV2` two types of actions exist:

- `addLiquidity`
- `removeLiquidity`

While the `removeLiquidity` logic correctly applies slippage checks after modifying liquidity, `addLiquidity` enforces slippage limits before the liquidity change is executed. It relies on estimated outputs from a resolver rather than on the actual asset deltas returned by the mint operation. This is contrary to Uniswap's warning:

"When pre-calculating liquidity changes, always account for rounding differences. Never assume getAmountsForLiquidity() == modifyLiquidity() deltas. Enforce slippage post-execution."

This discrepancy can cause the pre-execution slippage protection to fail in edge cases due to rounding or sync mismatches between estimated and actual deltas.

Impact:
Slippage protection for `addLiquidity` actions may be ineffective.

Recommendation:
Refactor the functions calling `_addLiquidity` to follow the same post-execution slippage pattern used in `_removeLiquidity`. Since the vault mint function returns the actual `amount0` and `amount1` used, forward these return values through `_addLiquidity` and compare them against `amount0Min` and `amount1Min` to enforce slippage after the liquidity change has occurred.

This aligns the implementation with Uniswap's guidance and ensures better slippage protection for end users.

Developer Response:
Acknowledged, we are rounding up amounts of token0 and token1 needed on the resolver smart contract, that should be enough for dealing with rounding differences.

L-02 - Minimum amount check on resolver makes one-sided liquidity not possible

Severity: Low Risk

Context:

- UniV4StandardModuleResolver.sol#L90-L92

Technical Details:
In the contract `UniV4StandardModuleResolver`, the function `getMintAmounts` includes a condition:

```
if (buffer >= maxAmount0_ || buffer >= maxAmount1_) revert
```

This check is intended to ensure that the maxAmount values are above a certain lower bound. However, the logic is overly restrictive: it reverts even when only one of the amounts is below the threshold, even if the other amount is valid.

This behavior contrasts with other parts of the codebase, such as the `addLiquidity` function in `ArrakisPublicVaultRouterV2`, which explicitly allows single-sided liquidity by permitting either `amount0Max` or `amount1Max` to be zero:

```
(params_.amount0Max == 0 && params_.amount1Max == 0)
```

Impact:
The current condition prevents valid single-sided liquidity provision. If either `maxAmount0_` or `maxAmount1_` is above the buffer while the other is below, the function will revert

unnecessarily. As a result, users cannot supply liquidity using just one token when it would otherwise be acceptable.

Recommendation:
Update the condition to only revert if both amounts are below the required threshold:

```
    if (buffer >= maxAmount0_ && buffer >= maxAmount1_) {
        revert MaxAmountsTooLow();
    }
```

Developer Response:

Acknowledged, `init0` and `init1` will be set in a way that this scenario should not occur.

L-03 - Missing event emission when updating the swap executor

Severity: Low Risk

Context:

- ArrakisPublicVaultRouterV2.sol#L116

Technical Details:

The `updateSwapExecutor` function does not emit an event whenever the swapper address is updated:

```
function updateSwapExecutor(
        address swapper_
    ) external whenNotPaused onlyOwner {
        if (swapper_ == address(0)) revert AddressZero();
        swapper = IRouterSwapExecutor(swapper_);
    }
```

Impact:

Off-chain indexers rely on emitted events to detect and track on-chain activity efficiently, therefore this action would not be tracked.

Recommendation:

Emit the state transition between the previous swapper and the new swapper:

```
function updateSwapExecutor(
        address swapper_
    ) external whenNotPaused onlyOwner {
        if (swapper_ == address(0)) revert AddressZero();
+       emit swappetUpdated(swapper, swapper_);
        swapper = IRouterSwapExecutor(swapper_);
    }
```

Developer Response:

Acknowledged.

L-04 - Integrators can be affected by lack of vault validation

Severity: Low Risk

Context:

- ArrakisPublicVaultRouterV2.sol#L895

Technical Details:

The `getMintAmounts` function allows anyone to view the shares they can mint from some max amounts:

```
 function getMintAmounts(
        address vault_,
        uint256 maxAmount0_,
        uint256 maxAmount1_
    )
        external
        view
        returns (
            uint256 shareToMint,
            uint256 amount0ToDeposit,
            uint256 amount1ToDeposit
        ){
        return _getMintAmounts(vault_, maxAmount0_, maxAmount1_);
    }
```

In the entire router contract it is always checked that the vault you are interacting with is whitelisted by the protocol. Otherwise the results can be faked using a phantom vault. In the `getMintAmounts` function, such vault address is not validated which could cause any smart contracts built on top of arrakis calling this function to be possibly vulnerable in case they are not validating the vault address to be whitelisted by arrakis.

Impact:

Integrators can be affected by lack of vault validation

Recommendation:

Add validation for the vault address:

```
function getMintAmounts(
        address vault_,
        uint256 maxAmount0_,
        uint256 maxAmount1_
    )
        external
        view
+       onlyPublicVault(vault_)
        returns (
            uint256 shareToMint,
            uint256 amount0ToDeposit,
            uint256 amount1ToDeposit
        ){
        return _getMintAmounts(vault_, maxAmount0_, maxAmount1_);
    }
```

Developer Response:
Acknowledged.

L-05 - `amountInSwap` can be bigger than `amountMax`

Severity: Low Risk

Context:

- ArrakisPublicVaultRouterV2.sol#L971
- ArrakisPublicVaultRouterV2.sol#L262

Technical Details:

When swapping, two tokens can be swapped, `token0` and `token1`, which they could also be `nativeToken`:

```
if (params_.swapData.zeroForOne) {
                if (token0_ != nativeToken) {
                    IERC20(token0_).safeTransfer(
                        address(swapper),
                        params_.swapData.amountInSwap
                    );
                } else {
                    valueToSend = params_.swapData.amountInSwap;
                }
            } else {
                if (token1_ != nativeToken) {
                    IERC20(token1_).safeTransfer(
                        address(swapper),
                        params_.swapData.amountInSwap
                    );
                } else {
                    valueToSend = params_.swapData.amountInSwap;
                }
            }
```

Whatever token amount will be swapped, it is referred to as `amountInSwap`. There is no check that makes sure that `amountInSwap` is smaller or equal than the deposited amounts `amount0Max` or `amount1Max`.

Impact: In case there is any dust in the contract from a previous transaction that was not reimbursed, it could be abused to forward such dust as part of the senders position.

Recommendation:

Do check in both cases, depending on what token is being swapped that the amount to be swapped is smaller or equal than the amount that was sent:

```
  if (params_.swapData.zeroForOne) {
+  if ( params_.swapData.amountInSwap > params_.addData.amount0Max ) revert
amountToSwapExceeded();
                if (token0_ != nativeToken) {
                    IERC20(token0_).safeTransfer(
                        address(swapper),
                        params_.swapData.amountInSwap
                    );
                } else {
                    valueToSend = params_.swapData.amountInSwap;
                }
            } else {
+  if ( params_.swapData.amountInSwap > params_.addData.amount1Max ) revert
amountToSwapExceeded();
                if (token1_ != nativeToken) {
                    IERC20(token1_).safeTransfer(
                        address(swapper),
                        params_.swapData.amountInSwap
                    );
                } else {
                    valueToSend = params_.swapData.amountInSwap;
                }
            }
```

Developer Response:
Acknowledged.

Informational

I-01 - Minor improvements to code and comments

Severity: Informational

Context:
See below.

Technical Details:

Fix the following:

- UniV4StandardModulePublic.sol#L43-L47 - Unused imports `SafeCast` and `SafeERC20`.
- UniV4StandardModulePublic.sol#L58-L59 - Unused libraries `BalanceDeltaLibrary` and `SafeERC20`.

Developer Response:

Acknowledged.

I-02 - Redundant deposit check

Severity: Informational

Context:

- UniV4StandardModulePublic.sol#L104

Technical Details:

The following check in the `deposit` function:

```
if (depositor_ == address(0)) revert AddressZero();
```

is redundant as the function is supposed to be called only from the meta vault:

```
  function deposit(
        address depositor_,
        uint256 proportion_
    ) external payable onlyMetaVault nonReentrant whenNotPaused returns
(uint256 amount0, uint256 amount1){
```

When checking how the vault calls the univ4 public module, it does forward `msg.sender` as the `depositor_` parameter in the `deposit` function:

```
bytes memory data = abi.encodeWithSelector(
            IArrakisLPModulePublic.deposit.selector,
            msg.sender,
            proportion_
```

Impact:

Redundant check.

Recommendation:

Remove the following check:

```
    function deposit(
        address depositor_,
        uint256 proportion_
    )
        external
        payable
        onlyMetaVault
        nonReentrant
        whenNotPaused
        returns (uint256 amount0, uint256 amount1)
    {
        // #region checks.

-        if (depositor_ == address(0)) revert AddressZero();
```

Developer Response:
Acknowledged, but we won't make changes.

I-03 - Consider re-structuring the pausing roles

Severity: Informational

Context:

- ArrakisPublicVaultRouterV2.sol#L104-L115

Technical Details:

The pausing roles could be improved for future integrations with on-chain monitoring providers and to avoid delayed pausing.

The ideal set-up is that more than one role is able to pause the contracts while only the owner can unpause them. The owner should also be able to grant or revoke the permissions of the addresses that can pause the contract. This is done as pausing is a time sensitive emergency mechanism where seconds matter. Additionally, if integrating with an on-chain monitoring provider where they automatically pause the contracts, they would need such permissions.

```
    function pause() external onlyOwner {
        _pause();
    }

    /// @notice function used to unpause the factory.
    /// @dev only callable by owner.
    function unpause() external onlyOwner {
        _unpause();
    }
```

Impact:

Possible delayed pausing mechanism and incompatible with on-chain monitoring mechanisms

Recommendation:

Update the permissions so that a more global pauser role is able to pause the contract if needed. In that pauser role, the Owner should also be included and be able to grant or revoke other addresses from that role:

```diff
- function pause() external onlyOwner {
+ function pause() external onlyPausers {
        _pause();
    }

    /// @notice function used to unpause the factory.
    /// @dev only callable by owner.
    function unpause() external onlyOwner {
        _unpause();
    }
```

Developer Response:

Acknowledged. We will implement a similar structure where the pauser will be a smart contract with different types of pausers. We can also differentiate pausers and unpausers addresses inside this Pauser smart contract.

Disclaimer