SHERLOCK

Security Review For Arrakis

Collaborative Audit Prepared For:	**Arrakis**
Lead Security Expert(s):	**0x73696d616f** **LZ_security**
Date Audited:	**June 17 - June 23, 2025**
Final Commit:	**04ecf6a**

Introduction

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

Scope

Repository: ArrakisFinance/arrakis-modular

Audited Commit: 26cfce9d8592fb5973547475378704c1d7b9b26a

Final Commit: 04ecf6aa58bc7ea5a0009b63d6124351cdcc402c

Files:

- src/ArrakisPublicVaultRouterV2.sol
- src/abstracts/UniV4StandardModule.sol
- src/libraries/UnderlyingV4.sol
- src/libraries/UniswapV4.sol
- src/modules/UniV4StandardModulePublic.sol
- src/modules/resolvers/UniV4StandardModuleResolver.sol

Final Commit Hash

04ecf6aa58bc7ea5a0009b63d6124351cdcc402c

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	3	4

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Using mulDivRoundingUp() for proportion may lead to DoS.

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/7

Summary

The use of mulDivRoundingUp() for proportion may result in the calculated amount0ToDeposit and amount1ToDeposit being greater than maxAmount0_ and maxAmount1_, leading to a DoS.

Vulnerability Detail

```
@>    uint256 proportion = FullMath.mulDivRoundingUp(
                shareToMint, BASE, totalSupply
            );
            (amount0ToDeposit, amount1ToDeposit) = UnderlyingV4
                .totalUnderlyingForMint(underlyingPayload, proportion);
```

The use of mulDivRoundingUp() may result in the calculated amount0ToDeposit and amount1ToDeposit being greater than maxAmount0_ and maxAmount1_, leading to a DoS.

This is particularly concerning within the totalUnderlyingForMint() function, where mulDivRoundingUp() is also employed.

Assuming totalSupply = 100e18, token0 = 100e18

(1) maxMount0 is 1e18 + 1, shareToMint = FullMath.mulDiv(maxMount0, totalSupply, token0) = 1e18 + 1
(2) proportion = FullMath.mulDivRoundingUp(shareToMint, BASE, totalSupply) = 1e16 + 1

(3) amount0 = FullMath.mulDivRoundingUp(proportion, token0, BASE) = 1e18 + 100
It can be seen that amount0 > maxMount0

POC:

```
function test_first_deposit_ETH_USDC()
    public {
        Currency currency0 = Currency.wrap(address(0));
        Currency currency1 = Currency.wrap(USDC);

        poolKey = PoolKey({
            currency0: currency0,
            currency1: currency1,
            fee: 10_000,
```

```
        tickSpacing: 10,
        hooks: IHooks(address(0))
    });

    IPoolManager(poolManager).unlock(abi.encode(0));


    // #region create a vault.

    bytes32 salt =
        keccak256(abi.encode("Public vault Univ4 salt v2"));
    init0 = 0;
    init1 = 1e18;
    maxSlippage = 10_000;

    bytes memory moduleCreationPayload = abi.encodeWithSelector(
        IUniV4StandardModule.initialize.selector,
        init0,
        init1,
        true,
        poolKey,
        IOracleWrapper(oracle),
        maxSlippage
    );

    bytes memory initManagementPayload = abi.encode(
        IOracleWrapper(oracle),
        TEN_PERCENT,
        uint256(60),
        executor,
        stratAnnouncer,
        maxSlippage
    );

    // #endregion create a vault.



    vm.prank(deployer);
    vault = IArrakisMetaVaultFactory(factory).deployPublicVault(
        salt,
        USDC,
        NATIVE_COIN,
        owner,
        uniswapStandardModuleBeacon,
        moduleCreationPayload,
        initManagementPayload
    );

    (uint256 sharesToMint, uint256 amount0, uint256 amount1) =
```

```
        IArrakisPublicVaultRouterV2(router).getMintAmounts(
            vault, 1, init1
        );

        console.log("sharesToMint is: ", sharesToMint);
        console.log("amount0 is: ", amount0);
        console.log("amount1 is: ", amount1);

        // #endregion deploy a vault.

        address  module = address(IArrakisMetaVault(vault).module());
        (uint256 init0, uint256 init1) =
↪   UniV4StandardModulePublic(payable(module)).getInits();
        console.log("init0 is: ", init0);
        console.log("init1 is: ", init1);


        address user = vm.addr(uint256(keccak256(abi.encode("User"))));

        deal( user, amount1 * 100);
        deal(USDC, user, amount0);

        // #region approve router.

        vm.startPrank(user);

        IERC20Metadata(USDC).approve(router, amount0);
        //IERC20Metadata(WETH).approve(router, amount1);

        // #endregion approve router.

        // #region add liquidity.

        IArrakisPublicVaultRouterV2(router).addLiquidity{value: amount1}(
            AddLiquidityData({
                amount0Max: 1,
                amount1Max: amount1,
                amount0Min: amount0 * 99 / 100,
                amount1Min: amount1 * 99 / 100,
                amountSharesMin: sharesToMint * 99 / 100,
                vault: vault,
                receiver: user
            })
        );
        //second add
        IArrakisPublicVaultRouterV2(router).addLiquidity{value: amount1*99}(
            AddLiquidityData({
                amount0Max: 1,
                amount1Max: amount1*99,
                amount0Min: amount0 * 99 / 100,
```

```
            amount1Min: amount1 * 99 / 100,
            amountSharesMin: sharesToMint * 99 / 100,
            vault: vault,
            receiver: user
        })
    );

    vm.stopPrank();

    uint256 getShareToken = IERC20(vault).balanceOf(user);
    console.log("user1 getShareToken: ", getShareToken);


    uint256 usdcOfmodule = IERC20(USDC).balanceOf(module);
    console.log("usdcOfmodule: ", usdcOfmodule);


     // #region second user deposit.
    uint256 user2MaxMount0 = 1;
    uint256 user2MaxMount1 = 1e18 +1;

    (sharesToMint, amount0, amount1) = IArrakisPublicVaultRouterV2(
        router
    ).getMintAmounts(vault, user2MaxMount0, user2MaxMount1);

    console.log("secondUser sharesToMint: ", sharesToMint);
    console.log("secondUser amount0: ", amount0);
    console.log("secondUser amount1: ", amount1);
    address secondUser =
        vm.addr(uint256(keccak256(abi.encode("Second User"))));

    deal(USDC, secondUser, user2MaxMount0);
    deal(secondUser, user2MaxMount1);
    // #region approve router.
    vm.startPrank(secondUser);

    IERC20Metadata(USDC).approve(router, user2MaxMount0);

    // #endregion approve router.

    IArrakisPublicVaultRouterV2(router).addLiquidity{value: amount1}(
        AddLiquidityData({
            amount0Max: user2MaxMount0,
            amount1Max: user2MaxMount1,
            amount0Min: amount0 * 99 / 100,
            amount1Min: amount1 * 99 / 100,
            amountSharesMin: sharesToMint * 99 / 100,
            vault: vault,
            receiver: secondUser
        })
```

```
        );

        vm.stopPrank();

        // #endregion second user deposit.
         uint256 secondUserShareToken = IERC20(vault).balanceOf(secondUser);
        console.log("secondUser getShareToken: ", secondUserShareToken);
        console.log(IERC20(vault).totalSupply());
        console.log("secondUser left USDC: ", IERC20(USDC).balanceOf(secondUser));
        console.log("secondUser left eth: ", secondUser.balance);
    }
```

result: revert. amount1 needed 10000000000000000100, is bigger than user2MaxMount1 = 1e18 +1

```
[FAIL: EvmError: Revert] test_first_deposit_ETH_USDC() (gas: 6590929)
Logs:
  sharesToMint is:  1000000000000000000
  amount0 is:  0
  amount1 is:  1000000000000000000
  init0 is:  0
  init1 is:  1000000000000000000
  user1 getShareToken:  99999999999999999000
  usdc0fmodule:  0
  secondUser sharesToMint:  1000000000000000001
  secondUser amount0:  0
  secondUser amount1:  1000000000000000100

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 2.65s (11.33ms CPU
↪  time)
```

Impact

May result in the calculated amount0ToDeposit and amount1ToDeposit being greater than maxAmount0_ and maxAmount1_, leading to a DoS. When it is not divisible, there is a very high probability of such a scenario occurring.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/5/files#diff-370c36cfec592d5c7726b7637db2b43e6db3745b9c8430cc0d43fe98c868b617R154-R158

Tool Used

Manual Review

Recommendation

1. Use Rounding Down when calculating proportion
2. Use the following formulas for calculating amount0ToDeposit and amount1ToDeposit:

```
uint256 amount0ToDeposit = FullMath.mulDivRoundingUp(
    shareToMint, currency0, totalSupply
);
uint256 amount1ToDeposit = FullMath.mulDivRoundingUp(
    shareToMint, currency1, totalSupply
);
```

3. Use Rounding Down when calculating liquidityDelta

The reason is this:

By rounding shareToMint down and amountsToDeposit up, we ensure that users always deposit more assets than the shares they receive (in USD terms).

The proportion is only used to calculate liquidityDelta, and when depositing, rounding down is definitely safe–at worst, it leaves a few extra tokens in the vault.

Discussion

lpetroulakis

Fixed by adding a check that that the `amount0ToDeposit/amount1ToDeposit` should be lower than `maxAmount0/maxAmount1`

Issue M-2: ArrakisPublicVaultRouterV2.addLiquidity Permit2 is missing a refund

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/11

Summary

ArrakisPublicVaultRouterV2.addLiquidityPermit2 is missing a refund

Vulnerability Detail

In addLiquidity, it is determined whether there are native tokens. If a user pays more ETH, the excess tokens will be returned to the user. But addLiquidityPermit2 did not do so. Therefore, if users add liquidity using addLiquidityPermit2, the excess ETH paid will not be refunded. Since amount0/amount1 is calculated, when the function is called, the user cannot accurately pay the correct amount of ETH. Therefore, the user may pay the excess ETH.

```
if (msg.value > 0) {
            if (token0 == nativeToken && msg.value > amount0) {
                payable(msg.sender).sendValue(msg.value - amount0);
            } else if (token1 == nativeToken && msg.value > amount1) {
                payable(msg.sender).sendValue(msg.value - amount1);
            }
        }
```

Impact

The excess ETH paid by the user has not been refunded

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/1/files#diff-4fe75457f540f4212d0e3434f6e782db14aa61e03184a0fc277d040998b7a8c3R321

Tool used

Manual Review

Recommendation

Add the judgment of refund just like addLiquidity

Issue M-3: Return amounts in UniswapV4::withdraw() are incorrect

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/16

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`UniswapV4::withdraw()` doesn't include the leftover amounts sent to the receiver in the returned amounts.

Vulnerability Detail

When withdrawing, the user receives:

1. Amounts from liquidity withdrawn;
2. Amounts from pending fees;
3. Amounts from amounts in the module that are not allocated.

Currently, the amounts sent to the receiver in 2 above are not considered on the return amount.

Impact

DoS when withdrawing funds as the return amounts will be lower than they really are.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/3/files#diff-b62a1f1e9b2a69f104cdd82e872a4f898b3fdd24f78a917236c9c98f3b52a9adR787

Tool Used

Manual Review

Recommendation

Add the leftover amounts to the return amount.

Discussion

Gevarist

We acknowledge this finding and it's a known issue, we don't consider it important enough to change the implementation.

Issue L-1: The same functionality is implemented using two different functions.

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/8

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The getAmountsForDelta() function serves the same purpose as the previously used LiquidityAmounts.getAmountsForLiquidity()

Vulnerability Detail

The getAmountsForDelta() function serves the same purpose as the previously used LiquidityAmounts.getAmountsForLiquidity(), but differs in rounding direction.

```
/// @notice Computes the token0 and token1 value for a given amount of liquidity,
↪  the current
/// pool prices and the prices at the tick boundaries
function getAmountsForDelta(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    int128 liquidity
) public pure returns (uint256 amount0, uint256 amount1) {
    if (sqrtRatioAX96 > sqrtRatioBX96) {
        (sqrtRatioAX96, sqrtRatioBX96) =
            (sqrtRatioBX96, sqrtRatioAX96);
    }

    if (sqrtRatioX96 < sqrtRatioAX96) {
        amount0 = SafeCast.toUint256(
            -SqrtPriceMath.getAmount0Delta(
                sqrtRatioAX96, sqrtRatioBX96, liquidity
            )
        );
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        amount0 = SafeCast.toUint256(
            -SqrtPriceMath.getAmount0Delta(
                sqrtRatioX96, sqrtRatioBX96, liquidity
            )
        );
        amount1 = SafeCast.toUint256(
            -SqrtPriceMath.getAmount1Delta(
                sqrtRatioAX96, sqrtRatioX96, liquidity
```

```
            )
        );
    } else {
        amount1 = SafeCast.toUint256(
            -SqrtPriceMath.getAmount1Delta(
                sqrtRatioAX96, sqrtRatioBX96, liquidity
            )
        );
    }
}
```

```
/// @notice Computes the token0 and token1 value for a given amount of liquidity,
↪  the current
    /// pool prices and the prices at the tick boundaries
    function getAmountsForLiquidity(
        uint160 sqrtRatioX96,
        uint160 sqrtRatioAX96,
        uint160 sqrtRatioBX96,
        uint128 liquidity
    ) internal pure returns (uint256 amount0, uint256 amount1) {
        if (sqrtRatioAX96 > sqrtRatioBX96)
            (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

        if (sqrtRatioX96 <= sqrtRatioAX96) {
            amount0 = getAmount0ForLiquidity(
                sqrtRatioAX96,
                sqrtRatioBX96,
                liquidity
            );
        } else if (sqrtRatioX96 < sqrtRatioBX96) {
            amount0 = getAmount0ForLiquidity(
                sqrtRatioX96,
                sqrtRatioBX96,
                liquidity
            );
            amount1 = getAmount1ForLiquidity(
                sqrtRatioAX96,
                sqrtRatioX96,
                liquidity
            );
        } else {
            amount1 = getAmount1ForLiquidity(
                sqrtRatioAX96,
                sqrtRatioBX96,
                liquidity
            );
        }
    }
```

Impact

The impact is Info. The code is difficult to maintain due to inconsistent implementations of the same functionality.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/6/files#diff-89e47e654e32ae326b49c87adf274367ef14fee0fdff05df6166df1a98913ccbR294

Tool Used

Manual Review

Recommendation

Use the same implementation function.

Discussion

Gevarist

We acknowledge that this code can be refactored; however, as these functions will not be modified, refactoring is not relevant for maintenance purposes at this time.

Issue L-2: It’s recommended to sync first the Pool Manager before settling even for currency0

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/14

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When settling currency0, the code currently doesn't sync, which is not recommended, but presents no issues currently.

Vulnerability Detail

The Uniswap Pool Manager indicates that even for currency0 `sync` should be called in order to avoid potential DoS.

Impact

There is no issue at the moment because none of the code flows trigger it, but it would be best to always sync first.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/3/files/6595beee6d8bb90ccb87691a4d1acd7eabdd8568#diff-b62a1f1e9b2a69f104cdd82e872a4f898b3fdd24f78a917236c9c98f3b52a9adR471

Tool Used

Manual Review

Recommendation

Always sync first.

Discussion

Gevarist

No impact here, we decide to not make fixes.

Issue L-3: UniswapV4::withdraw() doesn’t return the correct amount due to mixing liquidity to remove and fees accrued

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/17

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`UniswapV4:: _burnRanges()` only collects fees if liquidity to remove is > 0, which is wrong as the user may be entitled to a part of the fees even if they are not entitled to remove liquidity.

Vulnerability Detail

The user withdrawing is entitled to a proportion of liquidity and fees. The issue is that the code assumes that if the liquidity proportion is null, so is the fee proportion, which is not necessarily true. As a result they will receive slightly less, likely within 1-2 wei or similar amounts. This error is only noticeable if the proportion is very small and the fees exceed the liquidity in absolute value, so the damage is extremely limited.

A similar logical mistake occurs when depositing, but it never manifests there because of rounding up, which means that as long as proportion is > 0, there is always liquidity to withdraw. If proportion is 0, the user receives 0, hence no issue. If not having the liquidity rounding up, this could be problematic as the user would be able to deposit without paying their proportion in the fees accrued, but as it rounds up, it is not an issue when depositing.

Impact

User withdrawing a very small amount receives a few wei less.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/3/files#diff-b62a1f1e9b2a69f104cdd82e872a4f898b3fdd24f78a917236c9c98f3b52a9adR881

Tool Used

Manual Review

Recommendation

Discussion

Gevarist

We acknowledge this finding and it's a known issue, we don't consider it important enough to change the implementation.

Issue L-4: rebalanceResult is defined multiple times, and the first defined variable is not used.

Source: https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/issues/19

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`rebalanceResult` is defined multiple times, and the first defined variable is not used.

Vulnerability Detail

```
function rebalance(
        IUniV4StandardModule self,
        PoolKey memory poolKey_,
        IUniV4StandardModule.LiquidityRange[] memory liquidityRanges_,
        SwapPayload memory swapPayload_,
        IUniV4StandardModule.Range[] storage ranges_,
        mapping(bytes32 => bool) storage activeRanges_
    ) public returns (bytes memory result) {
        IPoolManager poolManager = self.poolManager();
        // #region fees computations.

@>        RebalanceResult memory rebalanceResult;

        // #endregion fees computations.

        {
@>            RebalanceResult memory rebalanceResult = _modifyLiquidity(
                poolManager,
                poolKey_,
                liquidityRanges_,
                swapPayload_,
                ranges_,
                activeRanges_
            );
        }
```

It can be seen that rebalanceResult is defined twice.

Impact

No serious impact for now.

Code Snippet

https://github.com/sherlock-audit/2025-06-arrakis-uniswap-v4-module-june-17th/pull/3/files#diff-b62a1f1e9b2a69f104cdd82e872a4f898b3fdd24f78a917236c9c98f3b52a9adR73

Tool Used

Manual Review

Recommendation

Remove the duplicate definition.

Discussion

Gevarist

We acknowledged this duplication.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.