Spectra Security Review

Pashov Audit Group

Conducted by: EV_om, unforgiven, 0xbepresent

February 24th 2024 - March 2nd 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **spectra-core** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Spectra

Spectra allows to split the yield generated by an Interest Bearing Token (IBT) from the principal asset. The IBT is deposited in the protocol and the user receives Principal Tokens (PT) and Yield Tokens (YT) in return. The PT represents the principal asset and the YT represents the yield generated by the IBT. Holders of the yield token for a specific IBT can claim the yield generated by the corresponding deposited IBTs during the time they hold the YT.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - fec59dc6720fb4861b07b30845ef2c1a42f947bf

fixes review commit hash - afaf6925be7713c89745cac17cc264b549591a06

Scope

The following smart contracts were in scope of the audit:

- `Registry`
- `factory/Factory`
- `router/Commands`
- `router/Dispatcher`
- `router/Router`
- `router/Constants`
- `router/util/RouterUtil`
- `libraries/CurvePoolUtil`

7. Executive Summary

Over the course of the security review, EV_om, unforgiven, 0xbepresent engaged with Spectra to review Spectra. In this period of time a total of **18** issues were uncovered.

Protocol Summary

Protocol Name	Spectra
Repository	https://github.com/perspectivefi/spectra-core
Date	February 24th 2024 - March 2nd 2024
Protocol Type	Interest rate derivatives protocol

Findings Count

Severity	**Amount**
Medium	3
Low	15
Total Findings	**18**

Summary of Findings

ID	Title	Severity	Status
[M-01]	TRANSFER_FROM_WITH_PERMIT command is vulnerable to DOS via frontrunning	Medium	Resolved
[M-02]	Code doesn't work with fee on transfer tokens	Medium	Resolved
[M-03]	Attack surface in the middle of the execute()	Medium	Resolved
[L-01]	The Factory.registry may be modified during the deployment	Low	Resolved
[L-02]	Dispatcher::_dispatchPreviewRate may incorrectly calculate the rate	Low	Resolved
[L-03]	Unused imports in several files	Low	Resolved
[L-04]	Off-by-one error in max number of iterations of getDx()	Low	Resolved
[L-05]	Inlined duplicate _resolveTokenValue logic	Low	Resolved
[L-06]	Unnecessary Ownable2StepUpgradeable inheritance in Router	Low	Resolved
[L-07]	Inefficient ERC20 allowance handling	Low	Resolved
[L-08]	Router.execute() does not need to be payable	Low	Resolved
[L-09]	Router.msgSender can be reset to 0 while executing commands	Low	Resolved
[L-10]	Calling Router.onFlashLoan() when msgSender is address(this)	Low	Resolved
[L-11]	Incomplete permission setting in	Low	Resolved

	deployAll() affects reward claiming		
[L-12]	Unnecessary usage of approve	Low	Resolved
[L-13]	In function spotExchangeRate() code should use CURVE_DECIMALS	Low	Resolved
[L-14]	_dispatch() should ignore the command when _resolveTokenValue() returns 0	Low	Resolved
[L-15]	Function _addInitialLiquidity() doesn't have slippage protection	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] `TRANSFER_FROM_WITH_PERMIT` command is vulnerable to DOS via frontrunning

Severity

Impact: Medium, because it allows for denial of service (DOS) attacks which can prevent the execution of legitimate transactions.

Likelihood: Medium, given that the exploit requires specific conditions to be met, such as observing and front-running transactions in the mempool.

Description

The `TRANSFER_FROM_WITH_PERMIT` command is intended to allow off-chain signed approvals to be used on-chain, saving gas and improving user experience.

However, including this command in a call to `execute()` will make the transaction susceptible to DOS via frontrunning. An attacker can observe the transaction in the mempool, extract the permit signature and values from the calldata and execute the permit before the original transaction is processed. This would consume the nonce associated with the user's permit and cause the original transaction to fail due to the now-invalid nonce.

This attack vector has been previously described in Permission Denied - The Story of an EIP that Sinned .

The following test demonstrates how the `attacker=address(1337)` front-runs the `owner` transaction and calls `permit`, resulting in the command execution being reverted:

```
// File: test/Router/RouterTest.t.sol:ContractRouterTest
    function testPermitFrontRun() public {
        uint256 amount = 10e18;
        uint256 privateKey = 0xABCD;
        address owner = vm.addr(privateKey);
        underlying.mint(owner, amount);
        //
        // 1. Owner signs to router
        vm.startPrank(owner);
        underlying.approve(address(principalToken), amount);
        uint256 shares = principalToken.deposit(amount, owner);

        bytes32 PERMIT_TYPEHASH = keccak256(
            "Permit(
              addressowner,
              addressspender,
              uint256value,
              uint256nonce,
              uint256deadline
            )"
        );
        uint256 deadline = block.timestamp + 10000;
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(
            privateKey,
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    principalToken.DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(PERMIT_TYPEHASH, owner, address
                          (router), shares, 0, deadline)
                    )
                )
            )
        );
        vm.stopPrank();
        //
        // 2. Attacker frontrun the caller, the nounce is incremented.
        vm.prank(address(1337));
        principalToken.permit(owner, address
          (router), shares, deadline, v, r, s);
        //
        // 3. Router executes the permit and it will be reverted
        vm.startPrank(owner);
        bytes memory commands = abi.encodePacked(bytes1(uint8
          (Commands.TRANSFER_FROM_WITH_PERMIT)));
        bytes[] memory inputs = new bytes[](1);
        inputs[0] = abi.encode(address
          (principalToken), shares, deadline, v, r, s);
        vm.expectRevert();
        router.execute(commands, inputs);
        vm.stopPrank();
    }
```

The user's execution will be reverted, but the user can re-call the router, forcing it to re-call the router and formulate the commands, thus spending in another transaction.

Recommendations

Consider implementing a mechanism to check if an allowance for the desired amount exists in case the call to `permit()` fails. A potential code adjustment could be:

```
try IERC20Permit(token).permit(msgSender, address
  (this), value, deadline, v, r, s) {
    // Permit executed successfully, proceed
} catch {
    // Check allowance to see if permit was already executed
    uint256 currentAllowance = IERC20(token).allowance(msgSender, address
      (this));
    if(currentAllowance < value) {
        revert("Permit failed and allowance insufficient");
    }
    // Otherwise, proceed as if permit was successful
}
```

This approach ensures that the transaction can proceed if the end state is as expected, even if the permit call itself fails due to front-running or other issues.

[M-02] Code doesn't work with fee on transfer tokens

Severity

Impact: High, core functionality would be broken.

Likelihood: Low, having fee-on-transfer token is a probable scenario.

Description

There are some places in the code that assume the ERC20 transfer function will transfer a specified `amount` and this is not true for tokens with fee. This will cause wrong calculation results in those cases. Some of the places where this issue happens:

1. In `_addInitialLiquidity()` code assumes it transfers `_initialLiquidityInIBT` tokens and tries to transfer them to the Curve pool.
2. In `_dispatchPreviewRate()` code assumes it would transfer `value` from the user.

Recommendations

Consider significant code modifications (managing actual balances) or prevent fee-on-transfer from appearing during code execution or acknowledge that these tokens will always have wrong calculation

[M-03] Attack surface in the middle of the execute()

Severity

Impact: High, user would lose approved tokens.

Likelihood: Low, execution flow should reach a malicious address which is hard to reach

Description

Users can execute multiple commands through Router contract's `execute()` function. These commands allow to transfer funds from `msg.sender` which is set as `msgSender` in the first `execute()` call. The issue is that if during the `execute()` execution, the execution reaches a malicious address, the attacker can perform a variety of actions leading to funds stolen.

The condition "execution flow reaches a malicious address" can happen in various ways, for example with hook functionality that some ERC20 tokens have(ERC777, ERC677 and ...), the execution will reach the `recipient` address that is defined in the commands.

1. The address can reenter the Router contract and execute arbitrary commands like transferring the initial transaction sender's funds.

```
User1 -> Router.execute() -> .... -> malicious address -> Router.execute
     () -> USDT.transferFrom(User1, malicious address, )
```

2. `onFlashLoan()` function doesn't check the caller and the operator and so it would be possible to call it from an arbitrary address with arbitrary data. Users can execute multiple commands through Router contract's `execute()` function. The malicious address can reenter the Router contract through `onFlashLoan()` function and steal the original `msgSender` funds.

```
User1 -> Router.execute() -> .... -> malicious address -> Router.onFlashLoan
     () -> Router.execute() -> USDT.transferFrom(User1, malicious address, )
```

3. Users can set approval of Router's tokens for malicious address by setting arbitrary address in the `execute()` function's command (for example using `DEPOSIT_ASSET_IN_IBT` command and setting malicious address as `ibt`'s value). The malicious address can transfer Router contract's balance and steal the original `msgSender` funds.

```
A.  Malicious_address -> Router.execute() -> USDT.approve
  (malicious address, uint256.max)
B.  User1 -> Router.execute() -> .... -> Malicious_address -> USDT.transferFrom
  (Router, malicious address, )
```

Recommendations

Each attack requires a separate defense:

1. Code should have a whitelist address and only allow them to reenter the `execute()` function. (Router and `msgSender` addresses)
2. Function `onFlashLoan()` should check the `msg.sender()` and also check the `operator` and make sure they have a legit value.
3. The Router and Dispatcher contract code should set allowance to 0 after the command executions or the code should only set approval for trusted addresses (Curve pools and PT pools).

8.2. Low Findings

[L-01] The `Factory.registry` may be modified during the deployment

The `Registry.sol` contract is utilized to store valid addresses used within the protocol. When a `principalToken` is created using the `Factory::deployPT` function, the new `principalToken` is registered in `Registry.sol` (code line `Factory#L80`):

```
File: Factory.sol
65:      function deployPT
  (address _ibt, uint256 _duration) public override returns (address pt) {
...
...
78:          pt = address(new BeaconProxy(ptBeacon, _data));
79:          emit PTDeployed(pt, msg.sender);
80:          IRegistry(registry).addPT(pt);
...
...
```

As long as the `principalToken` has not reached its maturity and it is a `PT` registered within `Registry.sol` (code lines `Factory#L104-L109`), a `curvePool` associated with that `principalToken` can be created using the `Factory::deployCurvePool` function:

```
File: Factory.sol
096:     function deployCurvePool(
097:         address _pt,
098:         CurvePoolParams calldata _curvePoolParams,
099:         uint256 _initialLiquidityInIBT
100:     ) public returns (address curvePool) {
101:         if (curveFactory == address(0)) {
102:             revert CurveFactoryNotSet();
103:         }
104:         if (!IRegistry(registry).isRegisteredPT(_pt)) {
105:             revert UnregisteredPT();
106:         }
107:         if (IPrincipalToken(_pt).maturity() < block.timestamp) {
108:             revert ExpiredPT();
109:         }
...
...
```

Additionally, the admin has the capability to change the `Factory.registry` using the `Factory::setRegistry` function. The issue arises when the

`Factory.registry` is changed, as this alteration prevents users who could still create a `curvePool` for their `principalToken` from deploying the `curvePool`. This is due to a validation within `Factory::deployCurvePool` that prohibits deploying a `curvePool` if the `principalToken` is not registered within the `Registry`. Consider the following scenario:

1. A user creates a `principalToken` with a maturity of 1 year but does not immediately create the associated `curvePool`. This new `PT` is stored within `Registry.sol`.
2. After 10 days, for some reason, the admin decides to change the `registry` using the `Factory::setRegistry` function.
3. Later, the user decides to create a `curvePool` associated with the `principalToken`. However, the `PT` is not found within the new `Registry`, causing the following validation to revert the transaction:

```
File: Factory.sol
    function deployCurvePool(
        address _pt,
        CurvePoolParams calldata _curvePoolParams,
        uint256 _initialLiquidityInIBT
    ) public returns (address curvePool) {
...
        if (!IRegistry(registry).isRegisteredPT(_pt)) {
            revert UnregisteredPT();
        }
...
```

This impacts the user who deployed the `PT`, especially if assets have already been deposited into the `PT`. It is not feasible to redeploy a new `PT` to facilitate the deployment of the `curvePool`, causing there to be no pool to provide liquidity to the `PT`.

The recommendation is to migrate the registered `PT`s to the `new registry` in the `Factory::setRegistry`.

[L-02] `Dispatcher::_dispatchPreviewRate` may incorrectly calculate the `rate`

The function `Dispatcher::_dispatchPreviewRate` is designed to simulate the execution of various commands and obtain the rate, defined as the amount of tokens returned at the end of the sequence of operations executed by the router per unit of token sent to the contract.

The issue arises when the commands `TRANSFER_FROM` or `TRANSFER_FROM_WITH_PERMIT` are used in the middle of a batch, causing a reset of `previousAmount`, which is then erroneously used in subsequent transactions. Consider the following scenario:

1. A user executes the `TRANSFER_FROM` command in `Dispatcher::_dispatchPreviewRate`, transferring `IBTs`, leaving the (Router::_previewRate#L103-L126) `_previewRate = (1 exchangeRate, 1e18 commandAmount)`.
2. The user executes the `DEPOSIT_IBT_IN_PT` command and obtains some `PT` at a specific rate, which leaves the rate at `_previewRate = (0.9 exchangeRate, 0.9e18 commandAmount)`.
3. The user executes the `TRANSFER_FROM` command again, transferring some `PTs`, resetting the `_previewRate = (0.9 exchangeRate, 1e18 commandAmount)`. Note the `commandAmount` increase.
4. The user executes the `CURVE_SWAP` command, and the new exchange may increase since the new calculation uses an incorrect `previous amount` because of the reset in `step 3`.

```
File: Dispatcher.sol
...
...
            } else {
                amountIn = _resolvePreviewTokenValue
  (amountIn, _previousAmount);
                exchangeRate = ICurvePool(pool).get_dy
  (i, j, amountIn).mulDiv(
                    RayMath.RAY_UNIT,
                    amountIn
                );
            }
            return (exchangeRate, 0);
```

This can lead to an inaccuracy in the calculation of the `rate` if the user improperly uses `TRANSFER_FROM`.

It is recommended to ensure that all `TRANSFER_FROM` commands are placed at the beginning of the command list.

[L-03] Unused imports in several files

Several files in scope include unused import statements. These unnecessary imports can lead to confusion, increase the contract's bytecode size unnecessarily, and impact gas costs for deployment.

The following imports are unused and can be removed: Factory.sol#L14
RouterUtil.sol#L8
RouterUtil.sol#L12-L14
RouterUtil.sol#L16
CurvePoolUtil.sol#L5

[L-04] Off-by-one error in max number of iterations of `getDx()`

In the `getDx()` function of the `CurvePoolUtil` library, there is a potential off-by-one error due to how the loop termination condition is checked. The loop is intended to run a maximum number of iterations defined by `MAX_ITERATIONS_BINSEARCH`. However, the condition `if (loops > MAX_ITERATIONS_BINSEARCH)` allows the loop to execute one more iteration than intended.

Consider adjusting the condition to `if (loops >= MAX_ITERATIONS_BINSEARCH)`. This adjustment will ensure that the loop runs for exactly `MAX_ITERATIONS_BINSEARCH` iterations at most, aligning with the intended logic.

[L-05] Inlined duplicate `_resolveTokenValue` logic

In `Dispatcher.sol#L87` , there is an inconsistency in how the token value is resolved for the `TRANSFER` command. Unlike other commands that utilize the `_resolveTokenValue` method to handle the `Constants.CONTRACT_BALANCE` special cases, the logic for the `TRANSFER` command is inlined, directly checking and substituting the value. This approach not only duplicates logic but also deviates from the established pattern of using helper methods for value resolution, potentially leading to maintenance issues and inconsistencies in how token values are handled across different commands.

Recommendation: Refactor the `TRANSFER` command execution logic within the `_dispatch` function to utilize the `_resolveTokenValue` method for resolving token values.

[L-06] Unnecessary `Ownable2StepUpgradeable` inheritance in `Router`

The `Router` contract inherits both from `Ownable2StepUpgradeable` and from `AccessManagedUpgradeable` via `Dispatcher`. It looks as if originally, `Ownable2StepUpgradeable` might have been used to manage ownership and access control. However, the current implementation of the contract does not grant any special privileges to the owner, rendering the inheritance from `Ownable2StepUpgradeable` and its associated initialization redundant.

This redundancy could lead to confusion and unnecessary complexity in the contract's codebase, potentially increasing the risk of errors in future development or maintenance. To streamline the contract and enhance clarity, remove the inheritance from `Ownable2StepUpgradeable` and eliminate the related initialization logic in the constructor.

[L-07] Inefficient ERC20 allowance handling

The below pattern is used in multiple instances to set an unlimited allowance in case the current value is lower than necessary:

```
uint256 allowance = IERC20(_token).allowance(address(this), _spender);
if (allowance < _value) {
        IERC20(_token).safeIncreaseAllowance(_spender, type(uint256).max - allowance);
}
```

Occurrences of this pattern can be found in: Dispatcher.sol#L272
Router.sol#L165
Factory.sol#L263

However, `safeIncreaseAllowance()` will simply poll the current allowance again and add it back to the value passed in the argument before calling `forceApprove()`:

```
function safeIncreaseAllowance
  (IERC20 token, address spender, uint256 value) internal {
        uint256 oldAllowance = token.allowance(address(this), spender);
        forceApprove(token, spender, oldAllowance + value);
}
```

Calling `forceApprove()` directly would be clearer and more gas efficient:

```
uint256 allowance = IERC20(_token).allowance(address(this), _spender);
if (allowance < _value) {
        IERC20(_token).forceApprove(_spender, type(uint256).max);
}
```

[L-08] `Router.execute()` does not need to be payable

The `execute` method in the `Router` contract is marked as `payable`. However, none of the commands involve the transfer of ether. This could lead to users mistakenly sending ETH with the transaction and losing it, or to confusion about the method's purpose.

Remove the payable modifier to reflect its actual use case and prevent unnecessary ether from being locked in the contract.

[L-09] `Router.msgSender` can be reset to 0 while executing commands

If a command in `execute()` performs a call to the original `msg.sender` which then reenters `execute()`, the `msgSender` variable will already be set at the beginning of the reentrant call but will be reset at the end of it. This means subsequent commands can be executed with the `msgSender` set to 0.

While this doesn't seem to be exploitable in a malicious way, it is recommended to ensure the `msgSender` variable always denotes the original `msg.sender` as intended in order to minimize potential attack vectors. A possible mitigation could look as follows:

```
diff --git a/src/router/Router.sol b/src/router/Router.sol
index c364804..d72a63d 100644
--- a/src/router/Router.sol
+++ b/src/router/Router.sol
@@
   -59,8 +59,10 @@ contract Router is Dispatcher, IRouter, Ownable2StepUpgradeable, IE

         // Relying on msg.sender is problematic as it changes during a flash
         // loan.
         // Thus, it's necessary to track who initiated the original Router
         // execution.
+        bool topLevel;
         if (msgSender == address(0)) {
             msgSender = msg.sender;
+            topLevel = true;
         }
         // loop through all given commands, execute them and pass along outputs
         // as defined
         for (uint256 commandIndex; commandIndex < numCommands; ) {
@@
   -73,7 +75,7 @@ contract Router is Dispatcher, IRouter, Ownable2StepUpgradeable, IER
                 commandIndex++;
             }
         }
-        if (msgSender == msg.sender)
+        if (msgSender == msg.sender && topLevel)
             // top-level reset
             msgSender = address(0);
     }
```

[L-10] Calling `Router.onFlashLoan()` when `msgSender` is `address(this)`

`onFlashLoan()` can be called directly, which allows executing commands with the `msgSender` variable set to `address(this)`.

While this doesn't seem to be exploitable in a malicious way, it is recommended to ensure the `msgSender` variable can only be set to the original caller in order to minimize potential attack vectors.

Recommendation: check that `msgSender != address(0)` in `onFlashLoan()`.

[L-11] Incomplete permission setting in `deployAll()` affects reward claiming

In `Factory.deployAll()`, not all permissions that are set in the `deployPT()` function are applied to the deployed PT. Specifically, the permissions related to rewards harvesting and rewards proxy setting are omitted. This oversight means that, while the PT deployment and Curve pool deployment proceed as expected, the ability to claim potential rewards is not properly configured at the time of deployment.

While these permissions can be set after the fact by a global admin, relying on post-deployment adjustments for essential permissions introduces unnecessary operational complexity.

Recommendation:

Ensure that all relevant permissions, including those for rewards harvesting and rewards proxy setting, are applied within the `deployAll` function to align with the permissions set in the `deployPT` function. E.g. in `deployAll()`, the code can set the target function role for `SetRewardsProxy()` and `ClaimRewards()` in PT contracts.

[L-12] Unnecessary usage of approve

In commands `Commands.CURVE_REMOVE_LIQUIDITY` and `Commands.CURVE_REMOVE_LIQUIDITY_ONE_COIN` in Dispatcher code calls `_ensureApproved(lpToken, pool, lps)` which isn't necessary.

[L-13] In function `spotExchangeRate()` code should use `CURVE_DECIMALS`

According to the code's comments, the exchange rate should be 18 decimal, so the code should convert the price decimal from `CURVE_DECIMALS` to `UNIT` (which is 18). The current implementation code assumes that these two values are equal and doesn't transform the decimals fully and uses `UNIT` instead of the `CURVE_DECIMALS`.

```
if (_i == 0 && _j == 1) {
            return Constants.UNIT.mulDiv(Constants.UNIT, ICurvePool
              (_curvePool).last_prices());
        } else if (_i == 1 && _j == 0) {
            return ICurvePool(_curvePool).last_prices();
        }
```

[L-14] `_dispatch()` should ignore the command when `_resolveTokenValue()` returns 0

Some ERC20 tokens or vaults or pools don't allow for 0 amount interaction. Function `_dispatch()` performs commands and it's called by `execute()` function. The issue is that `execute()` may have multiple commands and each command may use previous commands' results and the router's balance, so the value for some commands may result in 0 and there is no check in the code to ignore the command when the value is 0. So in actions like transfer, deposit, withdraw, and add liquidity code would try to interact with contracts with 0 amount and the code would revert, it would be best if the code ignores the command when the value is 0 and tries to avoid revert.

[L-15] Function `_addInitialLiquidity()` doesn't have slippage protection

Users can call `deployCurvePool()` and deploy Curve pool for PT pool and the code creates the pool and adds liquidity to the Curve pool based on the user-specified price. The issue is that the real token prices (ibt-PT) may change when the transaction is in the mempool and when the transaction is included in the blockchain, the code would create a pool that its price differs from the real price (user funds will be added as liquidity with wrong price) and so an attacker can steal user funds by back-running and performing a swap.

The code doesn't have slippage protection to avoid this issue. Consider allowing users to specify max slippage and then check the user-specified price with a real on-chain price to be in the range of the slippage.