Reservoir

1 Executive Summary
2 Scope
- 2.1 Objectives

3 System Overview
- 3.1 Router v6
- 3.2 General Purpose Modules
- 3.3 Exchange Modules
- 3.4 Misc

4 Design-related Security Considerations
- 4.1 Third-party Side-effects
- 4.2 Trustless/Stateless Design vs. Security of Funds
- 4.3 Generality vs. Specificity
- 4.4 Integrity of Reservoir Bundles and Frontrunning
- 4.5 Security and Safe Execution rely on the Bundle encoder

5 Findings
- 5.1 UniswapModule - `ethToExactOutput` never refunds any leftover `ETH` Critical
- 5.2 BaseExchangeModule - onERCxxx Callbacks can be used to perform arbitrary zero value calls in the name of the exchange module Critical
- 5.3 SeaportOrders - Insufficient mitigation of front-running for `matchOrder()` based "Seaport Approval Orders" Major
- 5.4 SeaportModule - unchecked returns Major
- 5.5 ExchangeModules - Buyers can generically evade fees via the ERC-721/ERC-1155 `onReceive` callback Major
- 5.6 UniswapModule - `ethToExactOutput` can be misused to swap `WETH` instead of `ETH` and leftover `WETH` will not be refunded Major
- 5.7 Router - `execute` should refund excess ETH Major
- 5.8 Exchange module - Unsafe defaults and lack of Documentation Major
- 5.9 ExchangeModules - Potentially questionable admin activity susceptible to front-running Major
- 5.10 LooksRareModule/SeaportModule - token might not be returned correctly if param `nft` does not match order params Medium
- 5.11 Excessive use of low-level calls bypassing the type-system and contract existence checks Medium
- 5.12 The fee system can be evaded Medium
- 5.13 General lack of Input Validation Medium
- 5.14 SeaportModule - Lack of input validation in `matchOrders()` Minor
- 5.15 LooksRareModule - explicitly check if token supports ERC-1155 Minor

Date	August 2022
Auditors	Christian Goll, Martin Ortner

1 Executive Summary

This report presents the results of our engagement with **Uneven Labs** to review **Reservoir Protocol**, a set of utility contracts for cross-exchange NFT aggregation.

The review was conducted over one and a half weeks, from **August 08, 2022,** to **August 24, 2022,** by **Martin Ortner** and **Christian Goll**. A total of 10 person-days were spent.

During the engagement, the assessment team reviewed the system's architecture, main routing components, and exchange modules. Critical issues regarding call authorization and the exchange module's accounting have been identified. In addition to implementation-related findings, the assessment team has identified architectural weaknesses.

Due to the time-boxed nature of this assessment and the amount and classes of findings reported, we conclude that it is very likely that more issues are present in the current revision of the code. It is, therefore, highly recommended to address the findings reported, followed by a thorough review of the next iteration before going live. Furthermore, a follow-up engagement is recommended to cover additional functionality in the exchange modules. For reference, the review team was unable to cover the ZeroExV4Module within the narrow time allocated for this review.

In particular, we would like to highlight the design-related security considerations and the surrounding critical and major issues regarding the management and security of user funds.

2 Scope

The client provided the following information alongside the engagement:

- Engagement Overview & Scope
- Contract Documentation

Our review focused on the commit hash 805e47cb420df0961b860d8505c38e325197b386. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Uneven Labs** team, we identified the following priorities for our review:

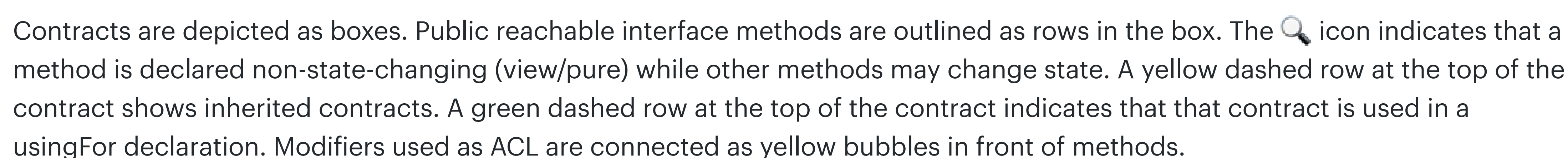
1. Ensure that the system is implemented consistently with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices and the Smart Contract Weakness Classification Registry.
3. Ensure the security of the flow of funds through the router and exchange modules.

The following list of contracts in order of priority has been provided by the development team ahead of the engagement:

- ReservoirV6_0_0
- SeaportModule
- UniswapV3Module
- LooksRareModule
- X2Y2Module
- ZeroExV4Module
- SeaportApprovalOrderZone
- PunksProxy
- FoundationModule
- BalanceAssertModule
- UnwrapWETHModule

3 System Overview

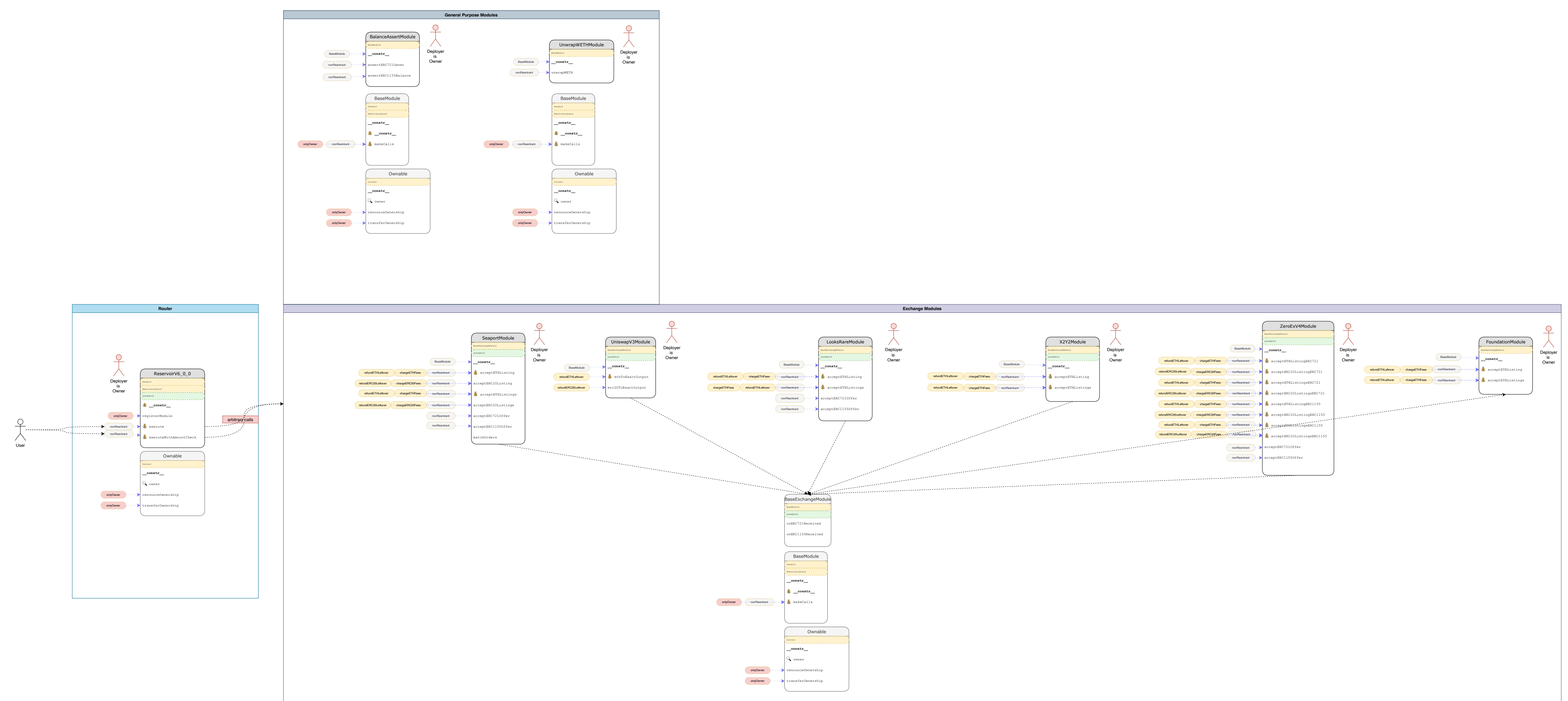
This section describes the top-level/deployable contracts, their inheritance structure, interfaces, actors, permissions, and essential contract interactions of the system under review.



Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The 🔍 icon indicates that a method is declared non-state-changing (view/pure) while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a usingFor declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.

5.16 LooksRareModule - `acceptERCxxxOffer` inconsistent use of argument names `takerBid` , `makerAsk` Minor

5.17 Where possible, a specific contract type should be used rather than `address` Minor



Router and Modules

The system comprises the `ReservoirV6_0_0` router contract and a set of exchange and general purpose modules.

3.1 Router v6

Is the main entry point that processes ExecutionInfo bundles with either the `execute()` or `executeWithAmountCheck()` functions.

The contract is ownable, allowing a privileged user to whitelist modules for use with execution info bundles. Execution info holds information about a low-level call (module address, call-data, call-value). The router resembles a multi-call facility. There was no information about how the privileged account is secured.

`execute()` executes transaction bundles and performs unchecked low-level calls to registered modules. An execution may or may not revert.

`executeWithAmountCheck()` extends `execute()` with low-level amount-checking capabilities. It performs an arbitrary `staticcall` to get an `uint256` from a defined contract before each execution-info in a bundle is executed, allowing the creation of a bundle with more executionInfos than necessary and returning early if `amountThreshold` is reached. The leftover `ETH` balance is refunded.

3.2 General Purpose Modules

UnwrapWETHModule

Unwraps the module's `WETH` balance and sends it to the `receiver`.

BalanceAssertModule

Provides assertions checking and reverting a transaction if the `ERC721` owner or `ERC1155` balance of a token contract does not match the provided criteria.

3.3 Exchange Modules

UniswapV3Module

The module is very generic, and the security highly relies on the parameterization of the method and swaps. This requires in-depth knowledge of the Uniswap protocol to avoid potential loss of funds.

- `ethToExactOutput` : allows to swap `ETH` or `WETH` to a target token via Uniswap `exactOutputSingle`. Swaps `ETH` / `WETH` balance of the exchange module contract balance. `ETH` is typically sent along from the router with a transaction to the exchange module for exchange. Leftover `ETH` is refunded to the address provided with an argument after the call. Note that this call does not refund leftover `WETH` !
- `erc20ToExactOutput` : allows to swap `ERC20` tokens. Assumes the exchange module is already in possession of the token to be swapped and does not pull in the token. Note that a token transfer must be performed as part of the atomic transaction bundle, or anyone can swap/spend them.

FoundationModule

Allows to buy NFT listings off foundation-protocol. The module provides two main entry points `acceptETHListing()` and `acceptETHListings()`. The latter is a convenience function that allows users to buy multiple listings in one transaction. The module calls foundation's buyV2 method that buys a listing at a set price. `msg.value` must be `<= maxPrice`, and any delta will be taken from the account's available `FETH` balance. Since the module is not supposed to hold any state, there should not be any `FETH` balance. Excess funds will be refunded to the exchange moodule and the exchange module `refundETHLeftover()` modifier returns the contract balance to the caller.

LooksRareModule

Allows to buy and sell `ERC721` and `ERC1155` via LooksRareExchange. It supports two types of fills:

- `matchAskWithTakerBidUsingETHAndWETH()` - matches an off-chain MakerAsk with an on-chain TakerBid. Taker buys `NFT` for `ERC20` according to Makers Ask. **fill buy-order** via the `acceptETHListing(s)` family of functions.
- `matchBidWithTakerAsk()` - matches an off-chain MakerBid with an on-chain TakerAsk. Taker accepts an off-chain offer to sell `NFT` for `ERC20` as outlined in Makers Ask. **fill sell-order** via the `acceptERCxxxOffer()` family of functions.

The trade may incur LooksRare protocol fees and royalties. Maker defines conditions and execution strategy of the trade. Taker is responsible for verifying that `minPercentageToAsk` (returns after fees) is what they agree with. The trade outcome highly depends on what the creator of the reservoir transaction bundle accepts or specifies in their taker/maker bids.

SeaportModule

Allows to fulfill and match orders on seaport.

- `matchOrders()` - can be called to match generic orders. There would be no validation if the executions were successful. It is not reentrancy protected by itself, likely due to it being a single forwarding call to `seaport.matchOrders()` and the function being reentrancy protected on the seaport side already.
- `accept[ERC20|ETH]listing()` - fill listings directly with `ETH` or `ERC20`. Excess funds are returned after the call and after taking fees. There is no check whether the order and the funds actually match. Internally calls `seaport.fulfillAdvancedOrder()`
- `accept[ERC20|ETH]listings()` - fills multiple listings directly with the same asset in `ETH` or `ERC20`. Excess funds are returned after the call and after taking fees. There is no check whether the order and the funds actually match. Internally this calls `seaport.fulfillAvailableAdvancedOrders()`.
- `accept[ERC721|ERC1155]Offer()` - attempts to fill a single offer to sell an `ERC721` / `ERC1155`. Returns the token to `params.refundTo` if the fill fails. Note that this method might fail to return the nft if `revertIfIncomplete == false` and Seaport performs a zero fill instead of reverting.

X2Y2Module

Allows users to buy only `ERC721` tokens off X2Y2 exchange. At present, only listings using ETH are supported.

- `acceptETHListing()` - fills a listing directly with ETH. Calls `buy()` internally.
- `acceptETHListings()` - convenience variant of the above that fills multiple listing in the same fashion as `acceptETHListing()` via a for-loop.
- `buy()` - Internal function that calls `x2y2_r1.run()`. Validates whether there's only one `SettleDetail` and `Pair[]` before calling `run()`. Transfers the `ERC721` token to the recipient directly from the module itself.

3.4 Misc

PunksProxy

Acts as an `ERC721` compliant wrapper for the CryptoPunks NFT collection considering they predate the standard. Implements all functions declared in the ERC721 specification such as `checkOnERC721Received()` with the wrapped functions calling `CryptoPunksMarket` internally. It should be noted that the Proxy allows transfers to the zero address.

SeaportApprovalOrderZone

Validates SeaportApprovalOrders using `tx.origin` in order to prevent bad actors from front-running one-time Seaport orders to the exchange modules.

- `isValidOrder()` - checks if `offerer` is `tx.origin` and reverts of that isn't the case. Returns a magic value indicating success.
- `isValidOrderIncludingExtraData()` - variant function of the above including `AdvancedOrder` and `CriteriaResolver` data.

4 Design-related Security Considerations

The system implements a set of smart contracts to provide a toolkit for a wide range of third-party systems. This entails a general gateway to NFT marketplaces and decentralized exchanges. The resulting interactions are bundled and executed against the main routing component. Such a generalized architecture must make decisions on the specificity of interaction with third-party components, the management of user funds, and potential side-effects resulting from external dependencies. These decisions not only impact the ease of integration but also the security impact on the overall smart contract system. This section summarizes the design issues we have identified.

4.1 Third-party Side-effects

Third-party providers can perform arbitrary actions in and around the system. This includes further external calls such as callbacks or nested multi-call functionality. Potential side-effects of external dependencies can be used to exploit the fact that Reservoir modules hold an intermediate token or ETH balance. External calls can cause state changes to be manifold, and implementing checks covering all unintended state changes may not be feasible.

4.2 Trustless/Stateless Design vs. Security of Funds

The Reservoir modules aim to carry as little state as possible to remain maintainable and avoid the concentration of risk, e.g., by holding token approvals. This poses a structural issue where the effort to keep the statefulness as low as possible results in workarounds that can potentially put funds at risk. An example of such occurrence is the design around the Seaport module, which sells to itself and transfers the token to circumvent the need for a user-given approval and thus a separate transaction. The specific scenarios outlined in the Findings below can put a user's tokens at risk.

Generally, users leveraging Reservoir for their trades may use it more than once, increasing the value of an extra transaction for approval when comparing it to the potential security trade-offs. The risk can further be reduced by centralizing approvals in a simple router with minimal attack surface, which then `delegatecall`'s into modules. This avoids the dispersion of approval addresses across individual module addresses, which is harder to manage.

4.3 Generality vs. Specificity

To reach as many third-party providers as possible, module contracts are used, which act as adapters to the external interfaces. Internally, they translate the target contracts to a more workable interface. This presents a design challenge where user flows need to be abstracted, but sometimes implementation details must be surfaced for user configuration.

In some cases, Reservoir uses low-level calls to avoid the issue of fully generalizing a component. Call data, value, and the target module are user-supplied with arbitrary values. While more straightforward to use programmatically, low-level calls come with various drawbacks, such as bypassing compile-time checks and the Solidity type system. Furthermore, user-supplied data can avoid authorization-related functionality, e.g., transferring ETH or tokens out of the module contract.

A more rigorous system specification and restrictions around protected admin functionality can solve this design issue.

It is essential to realize that the security of a generalized system can only be assured for the use-cases it was specified for. A lack of specification leaves room for interpretation and will eventually lead to functionality being used outside safe operation margins.

For example, the security of the system and execution bundles relies on the fact that execution bundles are constructed correctly according to the safe security boundaries the modules can be operated under. The specification should contain a description of the modules, the safe ways they can be utilized and combined with, module assumptions (i.e., tokens are sent via seaport orders, and the exchange module owns the tokens before it is executed), and security consideration for both, the modules and the overall system. At present, high-level and security documentation, as well as inline commentary, is sparse, leaving it up to the creativity of the caller to combine modules in ways they may not be safely operated under.

4.4 Integrity of Reservoir Bundles and Frontrunning

A Reservoir bundle consists of one or more `ExecutionInfo` structs, executed one by one. Each struct holds the call's target, data, and value. Bundles are loosely formed, and there are no protective measures around maintaining their integrity. Once a bundle is published to the mempool, everyone can see it. A frontrunner can extract individual transactions and decode and execute them. This can result in user bundles being only partially completed or failing altogether. In the worst case, this can result in the loss of funds. Specific attack scenarios are outlined in the Findings section below.

4.5 Security and Safe Execution rely on the Bundle encoder

Execution bundles are created by an SDK that is under development. The SDK is not in-scope of this review, but we would like to stress that a bundle's proper execution highly relies on the off-chain SDK producing a valid and secure sequence of inputs to the Reservoir Router. Given that the methods are generalized to the point of performing almost no input validation checks (recipients being non-zero addresses, input array sizes match, …) and them being called via low-level contract calls, there is potential for error where funds may theoretically be lost. It should also be noted that the off-chain bundle encoder is likely not written in native solidity and bears the risk of having operating system, language, or encoder-specific inconsistencies or bugs that may lead to the produced bundle not decoding correctly when run in the ethereum world. It is, therefore, suggested to at least simulate execution bundles before submitting them on-chain to validate that inputs execute correctly under a sandbox environment.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major 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.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 UniswapModule - `ethToExactOutput` never refunds any leftover `ETH` Critical

Description

There's a subtle difference between swapping token-for-token and eth-for-token with `ethToExactOutput`. In the token-for-token case, the method calculates the `amountIn` required and pulls it from the caller token balance. When using `ETH`, the caller has to provide `maxAmountIn` as `msg.value` to the swap call. The swap might cost up to `maxAmountIn` but may cost less. The excess `ETH` is not automatically returned and must be pulled from the contract manually via `router.refundETH()` in the same transaction the swap happened.

The current system does not pull excess `ETH` from the swap router. Hence, there is nothing to refund for the `refundETH` modifier. The extra funds remain with the swap router, and anyone can recover them by calling `refundETH` themselves.

Examples

code/packages/contracts/contracts/router/modules/exchanges/UniswapV3Module.sol:L30-L42

```
function ethToExactOutput(
    IUniswapV3Router.ExactOutputSingleParams calldata params,
    address refundTo
) external payable refundETHLeftover(refundTo) {
    if (params.tokenIn != weth) {
        revert WrongParams();
    }

    IUniswapV3Router(swapRouter).exactOutputSingle{value: msg.value}(
        params
    );
}
```

Recommendation

Call `swapRouter.refundETH()`. There are test cases covering `ethToExactOutput`. However, none seem to verify the refund case.

5.2 BaseExchangeModule - onERCxxx Callbacks can be used to perform arbitrary zero value calls in the name of the exchange module Critical

Description

The exchange modules inherit `BaseExchangeModule`, which in turn inherits `BaseModule`. `BaseExchangeModule` provides generic handling of `onERCxxxReceive` callbacks. These callbacks are unauthenticated and are not necessarily always being called within a token transfer transaction but may be misused and called directly by malicious actors to perform arbitrary zero-value calls to arbitrary addresses in the name of the module.

We understand that the idea of this set of contracts is that they are primarily state-less, which is not entirely true because specific contracts - namely exchange modules - may be part of 3rd party contract state, e.g., have token balances, allowance, special

permissions to interact with other components, etc. Providing a call-anything-anywhere (with zero eth value) primitive drastically increases the attack surface on the contract system. It may allow someone to steal tokens, change states in other contracts, and interfere with contract interaction by front-running legitimate transactions to alter states in other components.

Furthermore, the `BaseModule` provides admin-only functionality to `recover` funds locked in the contract.

code/packages/contracts/contracts/router/modules/BaseModule.sol:L27-L44

```
// To be able to recover anything that gets stucked by mistake in the module,
// we allow the owner to perform any arbitrary call. Since the goal is to be
// stateless, this should only happen in case of mistakes. In addition, this
// method is also useful for withdrawing any earned trading rewards.
function makeCalls(
    address[] calldata targets,
    bytes[] calldata data,
    uint256[] calldata values
) external payable onlyOwner nonReentrant {
    uint256 length = targets.length;
    for (uint256 i = 0; i < length; ) {
        makeCall(targets[i], data[i], values[i]);

        unchecked {
            ++i;
        }
    }
}
```

Note that the unauthenticated `onERCxxxReceived` provides almost the same capabilities as `makeCalls` (except for the ETH value being hardcoded to zero). This essentially allows anyone to spend the exchange module's contract allowance or token holdings, which are assumed only to be an admin functionality. Also, note the comment that `makeCalls` helps redeem trading rewards, while with `onERCxxxReceived`, anyone can potentially redeem them.

Offering a call-anything-anywhere primitive to an external entity is often a problem and should be avoided. Especially if the contract may hold state in other contracts (e.g. `LOOKS` trading rewards, etc.).

Fuel to fire this primitive can be exploited by a token receiver via token callbacks inside the execution of a transaction bundle. For example, if the bundle passes control flow to a potentially untrusted counterparty (i.e. a token receiver via the token `onReceive` callbacks) that party may be able to call back into the exchange modules `onERCxxxReceive()` callback (cannot be reentrancy protected) to perform arbitrary actions, including stealing tokens the exchange module is temporarily in possession of. Depending on the configuration of the bundle this attack may or may not be successful. However, this underlines the problem of allowing arbitrary calls.

Examples

- `onERCxxxReceived` methods allow arbitrary executions

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L205-L220

```
function onERC721Received(
    address, // operator,
    address, // from
    uint256, // tokenId,
    bytes calldata data
) external returns (bytes4) {
    if (data.length > 0) {
        (address target, bytes memory callData) = abi.decode(
            data,
            (address, bytes)
        );
        makeCall(target, callData, 0);
    }

    return this.onERC721Received.selector;
}
```

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L222-L238

```
function onERC1155Received(
    address, // operator
    address, // from
    uint256, // tokenId
    uint256, // amount
    bytes calldata data
) external returns (bytes4) {
    if (data.length > 0) {
        (address target, bytes memory callData) = abi.decode(
            data,
            (address, bytes)
        );
        makeCall(target, callData, 0);
    }

    return this.onERC1155Received.selector;
}
```

Recommendation

Break the call-anything-anywhere primitive and limit it to only the target and safe functions to be called as part of the callback. Implement a lock that checks whether you expect this function to be called (`inFlight` flag is set on the first interaction with your contract, and only if this flag is set can one interact with the `onERCxxxReceived` method if feasible). Consider guarding the method with a reentrancy guard (might need a different flag). Assess and document what methods and targets are reachable from this

callback. Consider that actors might not only call this method out-of-band, but the external entity might also act maliciously and call this method in unexpected ways/or multiple times). Harden the code against misuse of this function.

5.3 SeaportOrders - Insufficient mitigation of front-running for `matchOrder()` based "Seaport Approval Orders" Major

Description

The Readme outlines filling orders that require anything other than ETH can be tricky as token contracts require spending approvals to be able to pull-in tokens on someone else's behalf. Sending tokens directly must be done within the same transaction bundle as outside of that. Anyone would be able to spend them.

To address this problem, the protocol invented "Seaport Approval Orders" that effectively misuse SSeaport to initiate a token transfer from the user's seaport approval to a destination address by matching a specially crafted order. The problem with this order is that, by default, anyone observing the signed order in the mempool may be able to execute it before the actual reservoir execution bundle is processed. If someone could front-run the bundle's execution by matching the "Seaport approval order" directly with Seaport, the exchange would send the approved tokens to one of the exchange modules, and due to the state-less nature of these modules, anyone would be able to spend them.

To mitigate this issue, the protocol implements a custom seaport zone. The zone attribute of a signed seaport order is a contract callback address that Seaport calls to validate whether the order should be executed or not.

> - In all other cases, we use Seaport approval orders. A Seaport approval order is a short-lived order (in the range of minutes) that can send any tokens to a particular recipient free of charge. This still requires approval on the Seaport contract (or a specific Seaport conduit) rather than on the router. One issue, though, is that without any other mechanism for protection, these orders can be front-run (e.g., someone could listen for these orders in the mempool and then create an execution that fills the Seaport approval order and transfers any received funds to them). To overcome this limitation, all such orders should be associated with the `SeaportApprovalOrderZone` zone, which verifies that no one other than the original transaction sender (e.g. `tx.origin`) can trigger the filling of the approval order.

This `SeaportApprovalOrderZone` contract is quite simple and essentially only allows the `seaport.matchOrder()` call to succeed if the `tx.origin` is the actual `offerer` (usually the initiator of the bundle).

code/packages/contracts/contracts/router/misc/SeaportApprovalOrderZone.sol:L21-L32

```
function isValidOrder(
    bytes32,
    address,
    address offerer,
    bytes32
) external view returns (bytes4 validOrderMagicValue) {
    if (offerer != tx.origin) {
        revert Unauthorized();
    }

    validOrderMagicValue = this.isValidOrder.selector;
}
```

It should be noted that `tx.origin` is a notoriously bad design decision when used to authenticate transactions. This is because users can be tricked into interacting with a malicious contract. That contract may take over execution flow and redirect it to a target that authenticates the origin only. Since callback tokens are a thing, this attack surface extended a lot. For example, consider Alice sending an `ERC721` to Bob (attacker). As part of the transfer, standard `ERC721` contracts will perform a check if the recipient is a contract and ready to receive the token. This is done by the `ERC721` token contract calling a specific function on the recipient, which passes control to a pot. malicious onERC721Received() callback. This callback contract may then interact with another contract that authenticates the caller via `tx.origin`, only allowing them to impersonate the initial caller.

Adding the `tx.origin` check does not eliminate the original vulnerability but changes the requirement for successful exploitation. Initially, it was enough to "observe" the signed approval order and submit it before the bundle gets executed. In contrast, with the `tx.origin` check, direct interaction with the `offerer` of the order is required. This makes the attacker more complex and potentially less likely to be exploitable in many cases, as the attacker now must seemingly force the offerer into a race for the value of the order. However, that's not exactly the case.

Consider the following scenarios:

- Alice prepares and submits a bundle that includes a seaport approval order to an exchange module. The approval order is valid for 2-3 minutes, giving a potential attacker multiple blocks to execute the attack.
- Bob observes this. Front-runs the bundle to force the execution bundle to revert. The order data is still valid for a couple of blocks.
- Bob tricks Alice into transferring them an unrelated `ERC721` / `ERC1155` (there are probably ways to incentivize this a bit). this is somehow atomically exchanged, initiated by Alice.
- The `ERC721` / `ERC1155` `onReceive` callback passes execution flow control to Bob's malicious contract. That contract takes the previously blocked signed "Seaport Approval Order" and submits it to `seaport.matchOrder()`. `matchOrder()` calls back into the seaport zone to verify that the order can be filled. `SeaportApprovalOrderZone` checks that `tx.origin == Alice` holds, which is true, as they initiated the transaction. Seaport transfers tokens to the exchange module, and Bob spends them immediately before returning from the callback accepting the original NFT transfer.

Another scenario would be Bob tricking Alice into interacting with one of their malicious contracts directly front-running the bundle.

An inline exploitation scenario would involve Alice crafting an execution bundle that - at some point - gives Bob control of the transaction flow (i.e., because one module sends an NFT/ERC20withCallback to Bob) mid-execution of the bundle.

Given that there are multiple ways Bob can take control of the execution initiated by Alice via callbacks, it may become hard to argue that the `tx.origin` authentication sufficiently eliminates all risks. Security relies on Alice not falling into any pitfalls ordering

their execution bundle in a vulnerable way, having a vulnerable bundle configuration per se, or directly or indirectly interacting with potentially untrustworthy counterparts within or outside execution bundles while a seaport order is valid.

Note: It was noticed that all but one test case for seaport approvals do not specify a zone. This is an unsafe default, and given that other users may get inspired by test cases to infer how to use the system, this may lead to loss of funds.

5.4 SeaportModule - unchecked returns Major

Description

`SeaportModule` assumes that `matchOrders()` and `fulfillAdvancedOrder()` revert if an order cannot be fulfilled. However, the function signature and documentation suggest that the methods return arguments that indicate an error in the matching or fulfillment. These return values are not checked in the `SeaportModule`. This can theoretically lead to the module not reverting if the fulfillment was incomplete even though `revertIfIncomplete` was set, which will cause an execution bundle to continue with execution.

Examples

- `fulfillAdvancedOrder`

Documentation

> returns: fulfilled | bool | A boolean indicating whether the order has been successfully fulfilled.

code/packages/contracts/contracts/router/modules/exchanges/SeaportModule.sol:L197-L206

```
try
    ISeaport(exchange).fulfillAdvancedOrder{value: value}(
        order,
        criteriaResolvers,
        bytes32(0),
        receiver
    )
returns (bool fulfilled) {
    success = fulfilled;
} catch {}
```

However, the current seaport version always returns `true`. But this may change with an upgrade.

contracts/lib/OrderFulfiller.sol:L133

```
return true;
```

This can be problematic when the caller assumes seaport to revert but it does not and performs a zero fill instead. The `revertIfIncomplete` check in `acceptERC721Offer()` then skips refunding the token that still belongs to the exchange module.

code/packages/contracts/contracts/router/modules/exchanges/SeaportModule.sol:L132-L151

```
function acceptERC721Offer(
    ISeaport.AdvancedOrder calldata order,
    // Use `memory` instead of `calldata` to avoid `Stack too deep` errors
    ISeaport.CriteriaResolver[] memory criteriaResolvers,
    OfferParams calldata params,
    NFT calldata nft
) external nonReentrant {
    approveERC721IfNeeded(nft.token, exchange);
    fillSingleOrder(
        order,
        criteriaResolvers,
        params.fillTo,
        params.revertIfIncomplete,
        0
    );

    if (!params.revertIfIncomplete) {
        // Refund
        sendAllERC721(params.refundTo, nft.token, nft.id);
    }
```

`acceptERC1155Offer()`:

code/packages/contracts/contracts/router/modules/exchanges/SeaportModule.sol:L172-L175

```
if (!params.revertIfIncomplete) {
    // Refund
    sendAllERC1155(params.refundTo, nft.token, nft.id);
}
```

- `matchOrders`

Documentation

> returns: executions | struct Execution[] | An array of elements indicating the sequence of transfers performed to match the given orders.

Recommendation

It generally does not seem to be easy to validate seaport orders. For example, the seaport order validator may either revert or return zeroed-out values and attempt a zero-fill without throwing an error.

contracts/lib/OrderValidator.sol:L129-L130

```
// Assuming an invalid time and no revert, return zeroed out values.
return (bytes32(0), 0, 0);
```

// Assuming an invalid time and no revert, return zeroed-out values. return (bytes32(0), 0, 0);

contracts/lib/OrderValidator.sol:L170-L180

```
if (
    !_verifyOrderStatus(
        orderHash,
        orderStatus,
        false, // Allow partially used orders to be filled.
        revertOnInvalid
    )
) {
    // Assuming an invalid order status and no revert, return zero fill.
    return (orderHash, 0, 0);
}
```

This can theoretically lead to reservoir transaction bundles continuing with execution without correctly asserting that all trades were carried out successfully. For example, creating a bundle where the seaport module is invoked last is dangerous, as there is no additional call asserting that the order succeeded, the value was correctly transferred, and the outcome and state of the transaction bundle are correct. Therefore, it is suggested to (a) check all return values as an additional safeguard and (b) check that the expected outcome was reached and no unexpected side-effects occurred. The latter will probably require an extensive set of assertions to be provided on-chain and forcing that the last action in a bundle is always a set of assertions, or else there'll always be an inherent risk of this module being used outside for what it was specified for.

Due to the time-boxed nature of this review, there was no time to go deep on the seaport codebase to find corner cases where assumptions by the reservoir team would not hold. It is suggested to conduct an in-depth analysis and specify under which premises the current system and module operate.

5.5 ExchangeModules - Buyers can generically evade fees via the ERC-721/ERC-1155 `onReceive` callback Major

Description

Some methods charge a fee using the `chargeETHFees` modifier. This modifier takes a snapshot of the `ETH` balance before the action and diffs it to the balance after charging a fee on the consumed `ETH`. Finally, `refundETHLeftover` returns excess `ETH` to the specified account.

Here's an example using this pattern (there are multiple occurrences of this):

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L25-L37

```
function acceptETHListing(
    NFT calldata nft,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
    external
    payable
    nonReentrant
    refundETHLeftover(params.refundTo)
    chargeETHFees(fees, params.amount)
{
    buy(nft, params.fillTo, params.revertIfIncomplete, params.amount);
}
```

Note how `chargeETHFees` takes the `ETH` balance diff after `buy()` returns.

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L75-L84

```
modifier chargeETHFees(Fee[] calldata fees, uint256 amount) {
    uint256 balanceBefore = address(this).balance;

    _;

    uint256 balanceAfter = address(this).balance;

    uint256 length = fees.length;
    if (length > 0) {
        uint256 actualPaid = balanceBefore - balanceAfter;
```

And the `buy()` methods buys the token and then `safeTransfers()` it from the contract to the `recipient` which triggers the `recipient.onERC721Received()` callback (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L402-L403).

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L65-L92

```
function buy(
    NFT calldata nft,
    address receiver,
    bool revertIfIncomplete,
    uint256 value
) internal {
    bool success;
    try
        IFoundation(exchange).buyV2{value: value}(
            nft.token,
            nft.id,
            value,
            receiver
        )
    {
        IERC721(nft.token).safeTransferFrom(
            address(this),
            receiver,
            nft.id
        );

        success = true;
    } catch {}

    if (revertIfIncomplete && !success) {
        revert UnsuccessfulFill();
    }
}
```

The callback gives the `recipient` an opportunity to send enough `ETH` to the exchange module to "zero-out" `actualPaid` (`balanceAfter == balanceBefore`) in the `chargeETHFees()` modifier causing the fee calculation to return zero. The `ETH` sent to the module to clear the fee will subsequently be returned to the `refundTo` address with `refundETHLeftover()` , which takes the current address' balance.

Note that the exchange module allows fallback `ETH` via the `BaseModule.receive()` method, bypassing reentrancy protection.

code/packages/contracts/contracts/router/modules/BaseModule.sol:L22-L23

```
receive() external payable {}
```

Here's a potential stack of events for this attack:

```
acceptETHListing()
  -> modifier refundETHLeftover prologue: NOP - empty
    -> modifier chargeETHFees prologue: take balance snapshot
        -> function body:
               buy() ..
                  --!--> safeTransfer(nft to recipient)
                       --!--> recipient.onERCxxxReceived() callback: can now send ETH to exchange modules to clear out fee
    -> modifier chargeETHFees epilogue: charge fee based on balance diff (note that recipient may have restored the initial balance by n ETH transfer)
  -> modifier refundETHLeftover epilogue: return this.balance
```

5.6 UniswapModule - `ethToExactOutput` can be misused to swap `WETH` instead of `ETH` and leftover `WETH` will not be refunded Major

Description

Uniswaps `ethToExactOutput` does not distinguish between native `ETH` and `WETH` as an input token when swapping for another token. In both cases, the input token is set to `WETH` , but when paying for the swap, the Uniswap logic checks if `msg.value` is non-zero and either takes the caller approved `WETH` balance or auto-converts native transaction `ETH` to `WETH` and pulls-in that value.

contracts/base/PeripheryPayments.sol:L52-L62

```
function pay(
    address token,
    address payer,
    address recipient,
    uint256 value
) internal {
    if (token == WETH9 && address(this).balance >= value) {
        // pay with WETH9
        IWETH9(WETH9).deposit{value: value}(); // wrap only what is needed to pay
        IWETH9(WETH9).transfer(recipient, value);
    } else if (payer == address(this)) {
```

`ethToExactOutput` does not refund any token balances. Hence, excess `WETH` is not returned.

Furthermore, the method does not enforce that `msg.value` actually matches `params.amountInMaximum` .

Examples

code/packages/contracts/contracts/router/modules/exchanges/UniswapV3Module.sol:L30-L41

```
function ethToExactOutput(
    IUniswapV3Router.ExactOutputSingleParams calldata params,
    address refundTo
) external payable refundETHLeftover(refundTo) {
    if (params.tokenIn != weth) {
        revert WrongParams();
    }

    IUniswapV3Router(swapRouter).exactOutputSingle{value: msg.value}(
        params
    );
}
```

Recommendation

Require that `msg.value == amountInMaximum` to ensure this method can only be used for swapping `ETH`.

5.7 Router - `execute` should refund excess ETH Major

Description

`execute()` is payable and accepts `ETH` that can be used with calls to submodules to swap, fill orders, etc. No mechanism guarantees that (a) all transaction `ETH` is spent on calls to registered modules, and (b) no `ETH` is left in the router contract when `execute()` returns.

Note that the system is supposed to be as stateless as possible, implying that no excess `ETH` can stay in the contract and provided value must be consumed within the transaction. In case excess `ETH` is left in the contract after a bundle was executed, anyone can spend those funds (no `msg.value` checks).

Examples

code/packages/contracts/contracts/router/ReservoirV6_0_0.sol:L43-L56

```
function execute(ExecutionInfo[] calldata executionInfos)
    external
    payable
    nonReentrant
{
    uint256 length = executionInfos.length;
    for (uint256 i = 0; i < length; ) {
        executeInternal(executionInfos[i]);

        unchecked {
            ++i;
        }
    }
}
```

Recommendation

Check that the total amount of value sent to modules is `>= msg.value` (note that according to the client, some modules might send `ETH` to the router, and that's the reason for there being a `receive()` fallback function). Check if excess `ETH` is at the contract address after the bundle is executed and return surplus funds to `msg.sender`. Note that potential leftover token balances are not supported, and users need guidance on how to fail a bundle after checking that it is executed correctly.

Might also apply to `executeWithAmountCheck()`.

5.8 Exchange module - Unsafe defaults and lack of Documentation Major

Description

Exchange module functionality is meant to be called via transaction bundles through the router only. The router is a multicall facility that allows for the sequential calling of whitelisted modules with specific parameters. Various exchange module functionalities can be insecure when the functionality is called directly or just not as intended.

For example, the Uniswap module provides a method to swap ERC20 tokens.

code/packages/contracts/contracts/router/modules/exchanges/UniswapV3Module.sol:L43-L53

```
function erc20ToExactOutput(
    IUniswapV3Router.ExactOutputSingleParams calldata params,
    address refundTo
) external refundERC20Leftover(refundTo, params.tokenIn) {
    approveERC20IfNeeded(
        params.tokenIn,
        swapRouter,
        params.amountInMaximum
    );
    IUniswapV3Router(swapRouter).exactOutputSingle(params);
}
```

This function assumes that the exchange module is already in possession of the ERC20 token being swapped, or the call to Uniswap will fail. It does not pull in the token after approval by the caller, as this is exactly what the team was trying to avoid - utility contracts with user approvals. Instead, a seaport self-order is assumed to be used in a transaction bundle to transfer user ERC20 tokens to the exchange module. For this, the code that creates the transaction bundle must be aware of the exchange module address (see risks https://github.com/ConsenSysDiligence/reservoir-audit-2022-08/issues/4).

If someone misuses the method by using the exchange module directly, transferring the token amount to the contract in one transaction, and then immediately calling `erc20ToExactOutput` in another transaction, generic front-running bots would likely frontrun and steal the token balance.

Examples

code/packages/contracts/contracts/router/modules/exchanges/UniswapV3Module.sol:L43-L53

```
function erc20ToExactOutput(
    IUniswapV3Router.ExactOutputSingleParams calldata params,
    address refundTo
) external refundERC20Leftover(refundTo, params.tokenIn) {
    approveERC20IfNeeded(
        params.tokenIn,
        swapRouter,
        params.amountInMaximum
    );
    IUniswapV3Router(swapRouter).exactOutputSingle(params);
}
```

Recommendation

The recipe-style architecture of the system makes it hard to give good recommendations for avoiding accidental misuse. Especially when modules can be called individually without enforcing a safe path through the router. However, a safe path might not even exist due to the suggested statelessness of the system. The functionality is only safe to use if a transaction bundle was crafted that considers all individual security considerations. In this example, an ERC20 swap requires a seaport order to first put funds directly on the exchange modules address. A mechanism checks that the complete bundle is executed exactly as expected. We recommend to re-think the architecture, adding extensive documentation outlining how the components are supposed to be used, including typical recipes, bundles, and flows, as well as potentially falling back to a traditional pull-token style swap that always refunds excess tokens and restricting the methods to be only callable by the router.

5.9 ExchangeModules - Potentially questionable admin activity susceptible to front-running Major

Description

Exchange modules implement `BaseModule`, which is `Ownable` and provides functionality like

- `makeCalls` - perform arbitrary value or value-less calls
- `sendETH` - withdraw contract `ETH`

This admin functionality allows an admin to perform arbitrary actions on behalf of the contract at any time. However, similar functionality can be achieved by using regular non-privileged user-facing interfaces. This will likely always put an admin into a race with other users/bots/MEV. Hence, it is questionable if this functionality makes sense the way it is implemented.

Note: This would be more severe if exchange modules would be state-full, store value, or keep value in 3rd party contracts outside a single transaction bundle.

Note: The `nonReentrant` modifier luckily mitigates a potential issue where a rogue 3rd party component may call into the exchange module mid-transaction (i.e., when the exchange module interacts with it) to steal value temporarily transferred to the module.

Examples

The `sendETH` function

For example, the same primitive `sendETH` provided to an admin can be re-created with any un-authenticated method that is decorated `refundETHLeftover`.

- `sendETH`

code/packages/contracts/contracts/router/modules/BaseModule.sol:L47-L53

```
function sendETH(address to, uint256 amount) internal {
    (bool success, ) = payable(to).call{value: amount}("");
    if (!success) {
        revert UnsuccessfulPayment();
    }
}
```

- `refundETHLeftover`

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L56-L64

```
modifier refundETHLeftover(address refundTo) {
    _;

    uint256 leftover = address(this).balance;
    if (leftover > 0) {
        sendETH(refundTo, leftover);
    }
}
```

- Example: calling `acceptETHListings(nfts=[], prices=[], params=[x.refundTo=attacker] fee=[])` with empty arguments and `params.refundTo=attacker` will refund `address(this).balance` to `attacker`.

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L41-L61

```
function acceptETHListings(
    NFT[] calldata nfts,
    uint256[] calldata prices,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
    external
    payable
    nonReentrant
    refundETHLeftover(params.refundTo)
    chargeETHFees(fees, params.amount)
{
    uint256 length = nfts.length;
    for (uint256 i = 0; i < length; ) {
        buy(nfts[i], params.fillTo, params.revertIfIncomplete, prices[i]);

        unchecked {
            ++i;
        }
    }
}
```

The `makeCalls` function

`makeCalls` provides an admin interface to call any target with any calldata on behalf of the exchange contract. This is typically used to interact with 3rd party token contracts but may be used to perform any contract interaction.

As outlined in https://github.com/ConsenSysDiligence/reservoir-audit-2022-08/issues/7, the `onERCxxxReceived` callbacks can be used to emulate arbitrary calls (with zero ETH value) partially.

code/packages/contracts/contracts/router/modules/BaseModule.sol:L27-L53

```
// To be able to recover anything that gets stucked by mistake in the module,
// we allow the owner to perform any arbitrary call. Since the goal is to be
// stateless, this should only happen in case of mistakes. In addition, this
// method is also useful for withdrawing any earned trading rewards.
function makeCalls(
    address[] calldata targets,
    bytes[] calldata data,
    uint256[] calldata values
) external payable onlyOwner nonReentrant {
    uint256 length = targets.length;
    for (uint256 i = 0; i < length; ) {
        makeCall(targets[i], data[i], values[i]);

        unchecked {
            ++i;
        }
    }
}

// --- Helpers ---

function sendETH(address to, uint256 amount) internal {
    (bool success, ) = payable(to).call{value: amount}("");
    if (!success) {
        revert UnsuccessfulPayment();
    }
}
```

Recommendation

There's no straightforward recommendation that can be given at this point other than reconsidering the design of the system and its assumptions surrounding statelessness.

5.10 LooksRareModule/SeaportModule - token might not be returned correctly if param `nft` does not match order params Medium

Description

`acceptERCxxxOffer` assumes that the exchange model already possesses the `ERC721/1155` to be traded. The method may or may not revert if the call to the exchange fails. For example, the function continues execution if `revertIfIncomplete == false`, leading to the `ERC721/1155` still owned by the exchange module. It must be returned to the original owner specified with `params.refundTo`.

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L35-L39

```
struct OfferParams {
    address fillTo;
    address refundTo;
    bool revertIfIncomplete;
}
```

Since the exchange modules are supposed to be state-less, the contract needs to refund the token to the original owner via `sendAllERC721()`. Tokens are not allowed to remain at the exchange module address as anyone would be able to spend them.

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L115-L122

```
        if (params.revertIfIncomplete) {
            revert UnsuccessfulFill();
        } else {
            // Refund
            sendAllERC721(params.refundTo, nft.token, nft.id);
        }
    }
}
```

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L145-L152

```
        if (params.revertIfIncomplete) {
            revert UnsuccessfulFill();
        } else {
            // Refund
            sendAllERC1155(params.refundTo, nft.token, nft.id);
        }
    }
}
```

However, the method only returns the token specified with the `struct nft` argument. That information does not necessarily have to match the token specified with the order pair, i.e., at least a check for `nft.token==makerOrder.collection` and `nft.id==makerOrder.tokenId` is missing.

SeaportModule:

Instead of taking an `struct nft` an an input parameter that may not match the order it is suggested to take that information from the orders `ConsiderationItem` instead.

code/packages/contracts/contracts/router/modules/exchanges/SeaportModule.sol:L147-L151

```
if (!params.revertIfIncomplete) {
    // Refund
    sendAllERC721(params.refundTo, nft.token, nft.id);
}
```

code/packages/contracts/contracts/router/interfaces/ISeaport.sol:L34-L41

```
struct ConsiderationItem {
    ItemType itemType;
    address token;
    uint256 identifierOrCriteria;
    uint256 startAmount;
    uint256 endAmount;
    address recipient;
}
```

Recommendation

Remove the `nft` argument from the function. When not reverting, return the token at auction from `makerOrder.collection && makerOrder.tokenId`.

5.11 Excessive use of low-level calls bypassing the type-system and contract existence checks Medium

Description

The router calls arbitrary addresses with arbitrary data and value via the solidity low-level `address().call` and `address().staticcall` interfaces. These interfaces are untyped, less secure, bypass compile-time checks, and don't perform contract existence checks.

Examples

For example, an erroneous bundle executed via `executeWithAmountCheck()` with a bundle that accidentally sets or abi-decodes `checkContract` to `address(0x0)` will execute just fine as `staticcall` will return `success=true` even though no code was executed at the target address.

The same would theoretically be possible for module calls. However, the whitelisting feature currently mitigates that assuming that an admin was not accidentally whitelisting an address that has no code (might happen, there is no way to un-whitelist an entry, and adding an un-whitelisting option might open up a different set of security issues).

```
»  bytes memory data = new bytes(0)
»  (bool success, bytes memory result) = address(0).staticcall(data)
»  success
true
```

Note: In a lot of cases `abi.decode()` will luckily prevent the code from continuing if insufficient bytes were returned from a low-level external call. However, if more than enough bytes are returned excess bytes are ignored and the rest is interpreted as the target type allowing the contract to continue. It is not recommended to rely on this side-effect as a security measure.

Recommendation

Reconsider the system design and provide a standard interface for all modules to be shared. Provide interface types and use them accordingly (issue 5.17) as the interface type will perform contract existence checks on typed calls. Consider reducing the number of low-level calls as they generally increase the attack surface. Implement contract existence checks for low-level calls that typed calls cannot substitute.

5.12 The fee system can be evaded Medium

Description

A platform may choose to encode a transaction for the Reservoir router system that subtracts a fee for their services. Reservoir is meant to be a state-less set of utility contracts, allowing external platforms to encode transactions with their own conditions and fees. However, the fee system in the exchange modules is entirely optional.

An end-user might use a website/3rd party tool or platform to build a recipe of actions for use with the reservoir contract system. The website will prompt the user to sign and submit raw transaction data. The end-user might take that raw transaction data, decode it, change the fee recipient or set fees to zero, and then submit the transaction independently. No fees will be charged to the platform.

Examples

Multiple similar examples exist.

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L25-L37

```
function acceptETHListing(
    NFT calldata nft,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
    external
    payable
    nonReentrant
    refundETHLeftover(params.refundTo)
    chargeETHFees(fees, params.amount)
{
    buy(nft, params.fillTo, params.revertIfIncomplete, params.amount);
}
```

Recommendation

By design, there is no easy way to safeguard against this in a state-less manner as the contracts are purely a utility to the external party. With stateful contracts, an enrollment process and fee accounting there might be a way to avoid fee evasion. However, this is not in scope of the current system.

5.13 General lack of Input Validation Medium

Description

The system is a state-less set of utility contracts with a router that low-level sub-calls into whitelisted 'module'-contracts. The modules may have arbitrary interfaces and do not share an ABI. For transactions to be executed properly, transaction data must be constructed and encoded. Failure to do so may not result in the transaction reverting as the results execute on the input data.

The smart contract functionality is very generic, with almost no safeguards for misparametrization or accidental misuse, essentially shifting security guarantees to the off-chain component creating and encoding user transaction bundles. The client provides a default SDK that can be used to create such bundles with the project. This SDK is not in the scope of this review, and we highly recommend creating extensive documentation for the different use-cases, recipe flows, and security considerations.

Adding to the above, the PunksProxy has no zero address checks on transfers. This is mirrored in the original code to allow for burning tokens however, it should be enforced in the proxy to minimise user error.

Examples

This is only one example but multiple methods charge fees without a check that the fee- or refund-receiver is non-zero. Considering that off-chain encoding libraries may fail to encode data properly and return `0x0` or empty bytes for an encoded data structure, this may result in funds being sent to `address(0x0)` instead of a valid receiver.

code/packages/contracts/contracts/router/modules/exchanges/ZeroExV4Module.sol:L172-L191

```
function acceptETHListingsERC1155(
    IZeroExV4.ERC1155Order[] calldata orders,
    IZeroExV4.Signature[] calldata signatures,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
    external
    payable
    nonReentrant
    refundETHLeftover(params.refundTo)
    chargeETHFees(fees, params.amount)
{
    buyERC1155s(
        orders,
        signatures,
        params.fillTo,
        params.revertIfIncomplete,
        params.amount
    );
}
```

Likewise, `chargeETHFees` may be configured to 100% (or more) fees by a platform providing the encoding services as there are no checks for sane bounds.

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L89-L92

```
actualFee = (fees[i].amount * actualPaid) / amount;
if (actualFee > 0) {
    sendETH(fees[i].recipient, actualFee);
}
```

Recommendation

Off-chain transaction encoders may fail to produce properly encoded data. To safeguard from conflicting function arguments or inputs that may result in the loss of funds, it is suggested to implement input validation to catch the apparent errors.

5.14 SeaportModule - Lack of input validation in `matchOrders()` Minor

Description

There are no checks to ensure partial-fill or criteria based orders are not passed to the `matchOrders()` method. This is an issue because the method does not support criteria-based or partial filling of orders. Since the Module does not check the return value of `matchOrders()` , it could lead to execution continuing despite the fact that fulfilment was incomplete. The team has mentioned that Seaport approval orders are meant to be one-time fillable orders without any criteria. Regardless, this should be documented.

Recommendation

If validating the orders is not possible, the team should atleast document this fact mirroring the Seaport code base itself.

5.15 LooksRareModule - explicitly check if token supports ERC-1155 Minor

Description

Always explicitly check if the target token supports the interface you expect. For example, in the looksRareModule, the `buy()` function generically handles multiple token standards. It checks for `ERC721` or else assumes an `ERC1155` . This might be fine in most cases, as the `ERC1155` will probably fail (if no contract is deployed at the address; the contract does not support the interface). However, as token standards evolve, there might be another standard besides ERC721/ERC1155 that define the same interface but with slightly different characteristics that add side effects, and unexpected side effects is something that should be avoided with smart contract systems.

Note that the constant `erc1155Interface` is not used anywhere.

Examples

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L170-L186

```
    IERC165(makerAsk.collection).supportsInterface(erc721Interface)
) {
    IERC721(makerAsk.collection).safeTransferFrom(
        address(this),
        receiver,
        takerBid.tokenId
    );
} else {
    IERC1155(makerAsk.collection).safeTransferFrom(
        address(this),
        receiver,
        takerBid.tokenId,
        makerAsk.amount,
        ""
    );
}
```

Recommendation

Explicitly check for `supportsInterface(erc1155Interface)` and perform actions accordingly. Revert if a token interface is not supported.

5.16 LooksRareModule - `acceptERCxxxOffer` inconsistent use of argument names `takerBid` , `makerAsk` Minor

Description

The order argument names to `acceptERCxxxOffer(takerBid, makerAsk, ...)` are inconsistent with the third-party exchange interface `looksrareExchange.matchBidWithTakerAsk(takerAsk=takerBid, makerBid=makerAsk)` . Note how the argument names to `acceptERCxxxOffer()` , `takerBid & makerAsk` are passed to `matchBidWithTakerAsk()` which actually expects `takerAsk & makerBid` . This family of methods allows an on-chain taker to accept an off-chain bid to sell a specific token at a set price.

upstream interface

contracts/LooksRareExchange.sol:L303-L307

```
function matchBidWithTakerAsk(OrderTypes.TakerOrder calldata takerAsk, OrderTypes.MakerOrder calldata makerBid)
    external
    override
    nonReentrant
{
```

```
function matchBidWithTakerAsk(
    OrderTypes.TakerOrder calldata takerAsk,
    OrderTypes.MakerOrder calldata makerBid
) external override nonReentrant;
```

protocols implementation

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L96-L113

```
function acceptERC721Offer(
    ILooksRare.TakerOrder calldata takerBid,
    ILooksRare.MakerOrder calldata makerAsk,
    OfferParams calldata params,
    NFT calldata nft
) external nonReentrant {
    approveERC721IfNeeded(makerAsk.collection, erc721TransferManager);

    bool success;
    try ILooksRare(exchange).matchBidWithTakerAsk(takerBid, makerAsk) {
        IERC20(makerAsk.currency).safeTransfer(
            params.fillTo,
            IERC20(makerAsk.currency).balanceOf(address(this))
        );

        success = true;
    } catch {}
```

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L126-L132

```
function acceptERC1155Offer(
    ILooksRare.TakerOrder calldata takerBid,
    ILooksRare.MakerOrder calldata makerAsk,
    OfferParams calldata params,
    NFT calldata nft
) external nonReentrant {
    approveERC1155IfNeeded(makerAsk.collection, erc1155TransferManager);
```

Recommendation

Rename declaration for `acceptERCxxxOffer(takerBid, makerAsk, ...)` to `acceptERCxxxOffer(takerAsk, makerBid, ...)` to match the upstream interface.

5.17 Where possible, a specific contract type should be used rather than `address` Minor

Description

Rather than storing `address` types and then casting to the definitive contract type, it's better to use the best type available so the compiler can check for type safety. For all variables: state, local, and argument/returns declarations. Downcast to unchecked `address` types only when needed.

Examples

Multiple occurrences, some examples:

- declare `IERC721 token` in function arguments

code/packages/contracts/contracts/router/modules/BalanceAssertModule.sol:L26-L35

```
function assertERC721Owner(
    address token,
    uint256 tokenId,
    address owner
) external nonReentrant {
    address actualOwner = IERC721(token).ownerOf(tokenId);
    if (owner != actualOwner) {
        revert AssertFailed();
    }
}
```

- declare `IERC20 token` in function arguments

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L66-L73

```
modifier refundERC20Leftover(address refundTo, address token) {
    _;

    uint256 leftover = IERC20(token).balanceOf(address(this));
    if (leftover > 0) {
        IERC20(token).safeTransfer(refundTo, leftover);
    }
}
```

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L101-L106

```
modifier chargeERC20Fees(
    Fee[] calldata fees,
    address token,
    uint256 amount
) {
    uint256 balanceBefore = IERC20(token).balanceOf(address(this));
```

- declare `IWETH weth` and downcast to `IERC20(address(weth))` if needed

code/packages/contracts/contracts/router/modules/UnwrapWETHModule.sol:L16

```
address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

- multiple modules: declare `IFoundation exchange` instead of an unchecked `address` that is cast to the type whenever it is being used

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L16-L17

```
address public constant exchange =
    0xcDA72070E455bb31C7690a170224Ce43623d0B6f;
```

Recommendation

Where possible, use more specific types instead of `address`. This goes for parameter types as well as state variable types.

5.18 One step ownership transfer

Description

`BaseModule` and `ReservoirV6_0_0` use OpenZeppelin's Ownable contract, whose ownership transfer is a one-step process. Necessary address changes should be a two-step transfer process. This allows recovering from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible.

Examples

code/packages/contracts/contracts/router/ReservoirV6_0_0.sol:L11-L12

```
contract ReservoirV6_0_0 is Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;
```

code/packages/contracts/contracts/router/modules/BaseModule.sol:L8-L9

```
abstract contract BaseModule is Ownable, ReentrancyGuard {
    // --- Errors ---
```

Recommendation

Make use of an ownable contract that enables a two-step critical address change, i.e., the first transaction registers the new address, and the second transaction from the incumbent address updates the old address with the new one (i.e. claims ownership).

An example is boringCrypto's BoringOwnable

5.19 Explicitly define which modules can receive fallback `ETH` transfers

Description

`BaseModule` exports the fallback `receive()` function which allows gracious `ETH` value transfers. However, not all modules are supposed to receive funds via the fallback function. `LooksRare`, for example, does not refund excess `ETH` provided to the trade and expects the correct value instead.

code/packages/contracts/contracts/router/modules/BaseModule.sol:L22-L25

```
receive() external payable {}

// --- Owner ---
```

To reduce risk and provide a clearly defined interface, removing the `receive()` method from the `BaseModule` should be considered, and only implementing it in the high-level modules that have to receive `ETH` via fallback value transfers.

5.20 Input Validation - pot. Panic on Out of bounds array access

Description

Many functions take unchecked arguments. For example, some functions assume that arrays passed in as arguments are the same size. If they differ, an out-of-bounds access Panic might be thrown by the EVM (refunding remaining gas). However, issues like this are hard to debug and are wasting gas for something bound to revert when reaching the out-of-bounds access condition.

https://docs.soliditylang.org/en/develop/control-structures.html#panic-via-assert-and-error-via-require

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L41-L46

```
function acceptETHListings(
    NFT[] calldata nfts,
    uint256[] calldata prices,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
```

code/packages/contracts/contracts/router/modules/BaseModule.sol:L31-L35

```
function makeCalls(
    address[] calldata targets,
    bytes[] calldata data,
    uint256[] calldata values
) external payable onlyOwner nonReentrant {
```

Recommendation

Check that array sizes that are supposed to be of the same length match up and fail early if not, saving the caller gas. Check that addresses (recipient, refund, …) are not `address(0)`. Generally, improved input validation.

5.21 Use safe defaults for parameters: `params.revertIfIncomplete` vs. `params.continueIfIncomplete`

Description

Defensive coding suggests always falling back to safe defaults.

For example, `params.revertIfIncomplete = abi.decode(0x000000..0, (bool))` decodes to `bool false` which is logically equivalent to "don't revert if my action failed". This might hide errors or lead to unexpected outcomes where a bundle was supposed to fail. Still, the bundle continues execution because it was not explicitly configured to revert on error. By renaming `params.revertIfIncomplete` to `params.continueIfIncomplete`, it becomes more explicit that execution in a bundle is allowed to fail instead of falling back to this behavior by default.

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L25-L37

```
function acceptETHListing(
    NFT calldata nft,
    ETHListingParams calldata params,
    Fee[] calldata fees
)
    external
    payable
    nonReentrant
    refundETHLeftover(params.refundTo)
    chargeETHFees(fees, params.amount)
{
    buy(nft, params.fillTo, params.revertIfIncomplete, params.amount);
}
```

5.22 BaseExchangeModule - `chargeETHFees` should return early if no fees are given

Description

There is no reason to fetch the balance before and after if no fees will be charged. Consider returning early if `fees.length == 0`;

Examples

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L75-L99

```
modifier chargeETHFees(Fee[] calldata fees, uint256 amount) {
    uint256 balanceBefore = address(this).balance;

    _;

    uint256 balanceAfter = address(this).balance;

    uint256 length = fees.length;
    if (length > 0) {
        uint256 actualPaid = balanceBefore - balanceAfter;

        uint256 actualFee;
        for (uint256 i = 0; i < length; ) {
            // Adjust the fee to what was actually paid
            actualFee = (fees[i].amount * actualPaid) / amount;
            if (actualFee > 0) {
                sendETH(fees[i].recipient, actualFee);
            }

            unchecked {
                ++i;
            }
        }
    }
}
```

5.23 Try Catch - unnecessary success flag

Description

This is a pattern that is seen throughout the codebase. Instead of setting a `success = true` flag in the try-result body, the function could return and generically handle the `params.revertIfIncomplete` logic in the catch-body or as-is in the function epilogue instead. Reference: Solidity Docs

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L134-L151

```
bool success;
try ILooksRare(exchange).matchBidWithTakerAsk(takerBid, makerAsk) {
    IERC20(makerAsk.currency).safeTransfer(
        params.fillTo,
        IERC20(makerAsk.currency).balanceOf(address(this))
    );

    success = true;
} catch {}

if (!success) {
    if (params.revertIfIncomplete) {
        revert UnsuccessfulFill();
    } else {
        // Refund
        sendAllERC1155(params.refundTo, nft.token, nft.id);
    }
}
```

5.24 Router - Potentially unnecessary admin functionality

Description

Assuming that all components are state-less, value is never permanently transferred to one of the components in the system, and anyone can potentially take ownership of funds allocated to one of the components, it is questionable whether there is any security gain in whitelisting allowed modules. In the end, the off-chain transaction bundle encoder selects which modules are to be called. The router acts as an extended multi-call facility.

5.25 BaseExchangeModule - Avoid maximum approval of assets.

Description

Where possible, allowance should be restricted to only the amount needed to make the trade. Max approval of assets is an anti-pattern and can be especially dangerous in a system that depends upon other protocols in the event they are hacked.

Examples

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L162-L171

```
function approveERC20IfNeeded(
    address token,
    address spender,
    uint256 amount
) internal {
    uint256 allowance = IERC20(token).allowance(address(this), spender);
    if (allowance < amount) {
        IERC20(token).approve(spender, type(uint256).max);
    }
}
```

Recommendation

Restrict allowance to the value inserted in the `amount` parameter.

5.26 Insufficient event emission

Description

The system implements several privileged and state-changing functions. Despite this, they all lack event emission, critical for off-chain monitoring, especially when performing incident response.

Examples

Some examples of privileged or state-changing methods not emitting an event.

code/packages/contracts/contracts/router/ReservoirV6_0_0.sol:L30-L32

```
function registerModule(address module) external onlyOwner {
    modules[module] = true;
}
```

code/packages/contracts/contracts/router/modules/BaseModule.sol:L27-L44

```
// To be able to recover anything that gets stucked by mistake in the module,
// we allow the owner to perform any arbitrary call. Since the goal is to be
// stateless, this should only happen in case of mistakes. In addition, this
// method is also useful for withdrawing any earned trading rewards.
function makeCalls(
    address[] calldata targets,
    bytes[] calldata data,
    uint256[] calldata values
) external payable onlyOwner nonReentrant {
    uint256 length = targets.length;
    for (uint256 i = 0; i < length; ) {
        makeCall(targets[i], data[i], values[i]);

        unchecked {
            ++i;
        }
    }
}
```

Recommendation

Emit essential events such as in the above examples.

5.27 BaseExchangeModule - `chargeETHFees.balanceAfter` can be moved inside of the if-true-branch

Description

There is no reason to retrieve `balanceAfter` if `length==0`. Consider moving the `balanceAfter` line inside the `if` clause.

Examples

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L75-L84

```
modifier chargeETHFees(Fee[] calldata fees, uint256 amount) {
    uint256 balanceBefore = address(this).balance;

    _;

    uint256 balanceAfter = address(this).balance;

    uint256 length = fees.length;
    if (length > 0) {
        uint256 actualPaid = balanceBefore - balanceAfter;
```

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L110-L116

```
uint256 balanceAfter = IERC20(token).balanceOf(address(this));

uint256 length = fees.length;
if (length > 0) {
    uint256 actualPaid = balanceBefore - balanceAfter;

    uint256 actualFee;
```

5.28 Code Style - Naming Convention

Recommendation

The solidity language project recommends following a consistent coding style and naming convention for solidity source code. In particular, the coding style-guide recommends to name `constant` variables with the all-caps naming scheme. This makes it easier to recognize constant variables throughout the code-base. Additionally, it is suggested to prefix non-public methods with an underscore `_`, clearly marking them as `internal` methods.

Examples

Some examples:

code/packages/contracts/contracts/router/modules/exchanges/FoundationModule.sol:L16-L17

```
address public constant exchange =
    0xcDA72070E455bb31C7690a170224Ce43623d0B6f;
```

code/packages/contracts/contracts/router/modules/UnwrapWETHModule.sol:L16

```
address public constant weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

code/packages/contracts/contracts/router/modules/exchanges/LooksRareModule.sol:L23-L27

```
address public constant exchange =
    0x59728544B08AB483533076417FbBB2fD0B17CE3a;

address public constant erc721TransferManager =
    0xf42aa99F011A1fA7CDA90E5E98b277E306BcA83e;
```

Avoid using all-uppercase for struct names as per the naming convention they may be confused with constants:

code/packages/contracts/contracts/router/modules/exchanges/BaseExchangeModule.sol:L41-L44

```
struct NFT {
    address token;
    uint256 id;
}
```

5.29 Remove unused imports

Description

The following contracts are imported but not referenced in the source units:

- `IERC20, SafeERC20, IERC1155, IERC721`

code/packages/contracts/contracts/router/ReservoirV6_0_0.sol:L6-L9

```
import {IERC1155} from "@openzeppelin/contracts/token/ERC1155/IERC1155.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

code/packages/contracts/contracts/router/ReservoirV6_0_0.sol:L11-L13

```
contract ReservoirV6_0_0 is Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;
```

- `IERC20`

code/packages/contracts/contracts/router/modules/BalanceAssertModule.sol:L5

```
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

- `IERC20, SafeERC20` - in `x2y2Module, UniswapV3Module, SeaportModule`

code/packages/contracts/contracts/router/modules/exchanges/X2Y2Module.sol:L4

```
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

code/packages/contracts/contracts/router/modules/exchanges/X2Y2Module.sol:L15-L16

```
contract X2Y2Module is BaseExchangeModule {
    using SafeERC20 for IERC20;
```

code/packages/contracts/contracts/router/modules/exchanges/SeaportModule.sol:L18

```
using SafeERC20 for IERC20;
```

Recommendation

Check all imports and remove all unused/unreferenced and unnecessary imports.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
router/ReservoirV6_0_0.sol	`0cda22f7fee42323d697c0a64ea8afd6432cf869`
router/interfaces/ICryptoPunksMarket.sol	`8f5e86d3b54c525a63274a80aec0a4758fed6e07`
router/interfaces/IExchangeKind.sol	`427058fa93429896545bdc7013637487be56d31d`
router/interfaces/IFoundation.sol	`d4809e3909834cb6164dda35b67561751142b803`
router/interfaces/ILooksRare.sol	`bc01049b627f7e53b4b83e56e2b3d0384fe36895`
router/interfaces/ISeaport.sol	`9dda26c50a1c777c107760651cfc80ad5ce4f9d8`
router/interfaces/IUniswapV3Router.sol	`14708c594ba2d6b927381e7219cb7adc5f57a53a`
router/interfaces/IWETH.sol	`5e0a3c15d5799e17c7f80a0a34a123acd2348f91`
router/interfaces/IWyvernV23.sol	`11abe27c4da0ed4fa37a33dba572d45ab980f5a2`
router/interfaces/IX2Y2.sol	`b95adf8058232b815dd76046884405327b59f986`
router/interfaces/IZeroExV4.sol	`84c64d6596cc82d99175cc1e9300e1b442de18d4`
router/misc/PunksProxy.sol	`c3d1f05338c5473aa8053530f68e1379246ddf55`
router/misc/SeaportApprovalOrderZone.sol	`5a2f14e5b0b7ae1589318aa1800463c5e595b985`
router/modules/BalanceAssertModule.sol	`2b385242292a73ece594d2f8559fc3e449af9de4`
router/modules/BaseModule.sol	`a0dcaf5529eef703f50bdc2d267a6fc36201a3a5`
router/modules/UnwrapWETHModule.sol	`d84788859f3aeb16095fa6b283592699f50de400`
router/modules/exchanges/BaseExchangeModule.sol	`3ada08a17e65f7ae4fa3e145f00c322b13167cf2`
router/modules/exchanges/FoundationModule.sol	`56432e8d3a032b51280cc8025037df593aa2d34b`
router/modules/exchanges/LooksRareModule.sol	`7b99291b58f9d6ff617130824e032c481b4c3620`
router/modules/exchanges/SeaportModule.sol	`f2cfcfa7e242e3f741b7f072bf9aab46caa8bfec`
router/modules/exchanges/UniswapV3Module.sol	`1ef56bc46daa8e8bacc1f7aebbb1cc3deaacf081`
router/modules/exchanges/X2Y2Module.sol	`74b48bd7f590a2d8a553c853f2630ddc9b917fa9`
router/modules/exchanges/ZeroExV4Module.sol	`f91ed7d24d031e0841f49e4f6beb8d6bf107fe6b`

Appendix 2 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.