1inch Exchange

Date	September 2021
Auditors	Nicholas Ward, Dominik Muhs

1 Executive Summary

This report presents the results of our engagement with the **1inch** development team to review the fourth iteration of their **Aggregation Router** and its surrounding code.

The review was conducted by Nicholas Ward and Dominik Muhs over the course of 20 person-days between September 13th and September 24th, 2021.

2 Scope

Our review focused on the `AggregationRouterV4` contract and its dependencies at commit hash `0cdb810149b4750dbb3c857f3dabee794c313ca9`. The `UnoswapRouter` was established as a low priority despite being a dependency of `AggregationRouterV4`.

The `AggregationExecutor` contract and its extensions and dependencies were explicitly excluded from the scope of the review. The `discountedSwap()` method of `AggregationRouterV4`, intended for use with pre-London EVM versions, was also excluded.

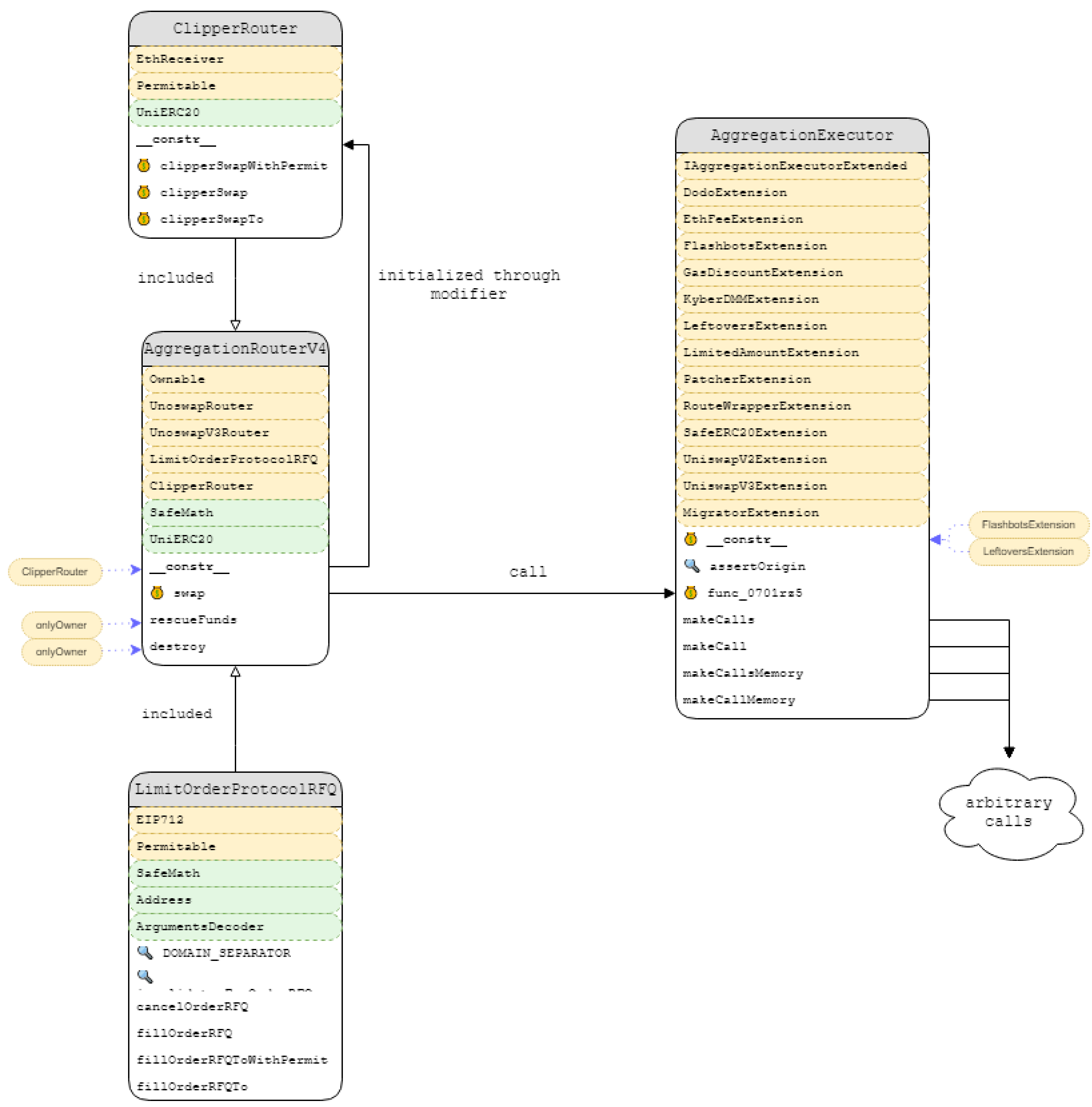
The complete list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the 1inch development team, we identified the following priorities for our review:

1. Ensure that user approvals cannot be drained from `AggregationRouterV4`.
2. Ensure that swaps through the router cannot violate the minimum return condition without failing.
3. Ensure that the system is implemented consistently with the intended functionality and without unintended edge cases.
4. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview



Out of scope components have been removed

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties or assumptions that were used as a basis for the review.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- Users
  - swap ETH, WETH, and tokens through the executor contract
  - swap ETH, WETH, and tokens through the Clipper trading interface
- Market makers
  - provide liquidity for takers via 1inch RFQ orders or external sources such as AMMs and other DEXs
  - interface with off-chain infrastructure to facilitate trades
- Owner
  - remove ETH and tokens from the Aggregation Router
  - self-destruct the Aggregation Router

4.2 Trust Assumptions

In any system, it is essential to identify what trust is expected/required between various actors.

The `AggregationRouterV4.swap()` function requires a user to submit the address and call data of the currently deployed `AggregationExecutor` contract. As the contract is closed-source, it becomes infeasible for a user to validate the contents of their transaction. There is an inherent trust in the executing medium, e.g. the 1inch frontend, to provide the correct call values. No reentrancy guards have been used across the system for gas-optimization purposes. However, it must be noted that an attacker, e.g., manipulating the front end, can provide malicious `swap` parameters, resulting in unsuspecting users executing arbitrary calls.

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 Malicious maker can satisfy balance delta check via other sources Critical

Resolution
The development team has addressed this issue in commit `0e667047e38bdd70be701205bc93c81eafe14194`. This change has not been reviewed by the audit team.

Description

The `AggregationRouterV4` contract relies on a check comparing the taker's `spentAmount` of the input token to the amount of output token received. If the taker's balance in the output token (`dstToken`) is increased by at least the specified `minReturnAmount`, the trade is considered successfully executed. For partial fills, this `minReturnAmount` is adjusted based on the amount of input token (`srcToken`) consumed.

This check not only protects takers from unexpected slippage, but it also serves as a critical safety check against a potentially malicious `AggregationExecutor`. Because the `AggregationExecutor` contract is outside the scope of this review and the source code will not be made public, this check is considered particularly important.

code/contracts/AggregationRouterV4.sol:L132-L137

```
if (flags & _PARTIAL_FILL != 0) {
    spentAmount = initialSrcBalance.add(desc.amount).sub(srcToken.uniBalanceOf(msg.sender));
    require(returnAmount.mul(desc.amount) >= desc.minReturnAmount.mul(spentAmount), "Return amount is not enough");
} else {
    require(returnAmount >= desc.minReturnAmount, "Return amount is not enough");
}
```

Exploit

This loose definition of a successful swap can be exploited by a malicious `AggregationExecutor` or any counterparty able to gain control of execution during the swap. Because of the black box treatment of the `AggregationExecutor` contracts, it is assumed that there could be many ways to take this control of execution. But, consider an ERC-777 callback from one of the traded tokens as a simple example.

In addition to control of execution, a potential attacker would need a mechanism to accomplish one of the following:

1. Increase the taker's balance in the output token in a way that does not decrease their own balance by the same amount.
2. Force unexpected side effects within 1inch or an external smart contract system.

Three concrete variations of this attack follow, all of which would occur between the initial check of the taker's `dstToken` balance and the final check on the success of the swap.

1. Force the taker to recognize the transfer of tokens as a deposit. If, for example, the taker was a DeFi protocol or DAO that also allowed shares to be purchased for the `dstToken`, the maker or `AggregationExecutor` could buy shares during the execution of the trade. The taker's balance would increase by the necessary amount to make the trade succeed, but the attacker could later withdraw the `dstToken` that the taker expected to receive outright.
2. Execute a limit order signed by the taker for the same `dstToken`. The increase in the taker's balance provided from the successful limit order execution would incorrectly satisfy the balance delta check for both swaps. For partial fill orders, execution of a limit order from the taker for the `srcToken` could be similarly exploited.
3. Force a payout of `dstToken` that is already owed to the taker from another smart contract. Take for example a user swapping `DAI` for `UNI` that also has an unclaimed payout in the Uniswap `MerkleDistributor` contract. The attacker could call `MerkleDistributor.claim()` with the necessary merkle proof, then force the transfer of `UNI` from the swap execution to fail.

   Assuming the amount of claimed `UNI` was at least the `minReturnAmount`, this would satisfy the final check on the taker's balance and the maker would receive the `DAI` from the trade. However, the maker would not have forfeited their `UNI`.

Recommendation

Use an intermediate address with approvals to the Aggregation Router to collect the tokens from a swap. Perform the balance delta checks on this intermediate address, then transfer the tokens to the intended destinations.

5.2 Potential memory corruption in `UnoswapRouter` Major

Resolution

The development team has addressed this issue in commit `c9d4bee4ac0fb6bb9d2da44278274fa0d40f69f8`. This change has not been reviewed by the audit team.

Description

The `UnoswapRouter` contract uses inline assembly to optimize swap interactions with UniswapV2 pools. Before performing a swap with a pool, it makes a staticcall to the following function on the caller-provided pool address:

```
getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTimestampLast)
```

This call specifies the return offset and expected return data length for the `staticcall`, expecting the memory indicated by this offset and length to be overwritten with the return data.

code/contracts/UnoswapRouter.sol:L55-L61

```
mstore(emptyPtr, _UNISWAP_PAIR_RESERVES_CALL_SELECTOR_32)
if iszero(staticcall(gas(), pair, emptyPtr, 0x4, emptyPtr, 0x40)) {
    reRevert()
}

let reserve0 := mload(emptyPtr)
let reserve1 := mload(add(emptyPtr, 0x20))
```

Note that while the `getReserves()` function is expected to return 96 bytes of data, the `UnoswapRouter` specifies the length to copy as 64 bytes, meaning the value returned for `_blockTimestampLast` is never copied into memory.

There are cases where a `staticcall` to `getReserves()` can succeed without returning the expected amount of data. Notably, this occurs for any call to an address that contains no code (e.g. a yet-to-be-deployed pair address). When this happens, the memory region passed to `staticcall` for return data is left unchanged. This means that if a pool address is either an account containing no code or a contract that returns less than 64 bytes of data, the `UnoswapRouter` will interpret dirty memory regions as `reserve0` and `reserve1`. While the `emptyPtr` from which this memory is read is manually updated based on the free memory pointer stored by Solidity, the memory beyond this free memory pointer can not be expected to be empty, as per the Solidity documentation.

Because the `UnoswapRouter` was heavily de-prioritized, the potential impact of this issue was not deeply explored. However, memory corruption issues of any kind should be considered high severity.

Recommendation

Either check that the `staticcall` returns the expected 96 bytes of data using `RETURNDATASIZE`, or replace the implicit memory copy by the `STATICCALL` opcode with an explicit `RETURNDATACOPY`, which will write zeros to the specified memory region if the requested copy length exceeds the length of the return data.

5.3 Dangerous use of inline assembly Medium

Resolution

The development team has addressed this issue in commit `4736ce74afaf41b97195b19d091de91fa62ef234`. This change has not been reviewed by the audit team.

Description

Yes, inline assembly can reduce gas costs compared to vanilla Solidity, sometimes significantly. However, it is crucial to consider, from a security perspective, *where* assembly is being used, *how* it is being used, and whether the gas saved is worth the risk introduced.

The 1inch development team is well aware of the risks of using assembly, and the intended use of the contracts in question does demand efficient resource consumption. This being said, our opinion is that the *where* and the *how* of this assembly usage should be reconsidered.

As an example, the `LimitOrderProtocolRFQ` contract makes heavy use of assembly in validating signed orders. It also uses assembly to modify caller-provided data in place before sending this data to a caller-provided address. This assembly is hidden behind layers of internal function calls, hindering both readability and auditability.

code/contracts/LimitOrderProtocolRFQ.sol:L172-L182

```
function _callMakerAssetTransferFrom(address makerAsset, bytes memory makerAssetData, address taker, uint256 makingAmount) private {
    // Patch receiver or validate private order
    address orderTakerAddress = makerAssetData.decodeAddress(_TO_INDEX);
    if (orderTakerAddress != address(0)) {
        require(orderTakerAddress == msg.sender, "LOP: private order");
    }
    if (orderTakerAddress != taker) {
        makerAssetData.patchAddress(_TO_INDEX, taker);
    }
    _makeCall(makerAsset, makerAssetData, makingAmount);
}
```

code/contracts/helpers/ArgumentsDecoder.sol:L32-L36

```
function patchAddress(bytes memory data, uint256 argumentIndex, address account) internal pure {
    assembly { // solhint-disable-line no-inline-assembly
        mstore(add(add(data, 0x24), mul(argumentIndex, 0x20)), account)
    }
}
```

code/contracts/LimitOrderProtocolRFQ.sol:L190-L196

```
function _makeCall(address asset, bytes memory assetData, uint256 amount) private {
    assetData.patchUint256(_AMOUNT_INDEX, amount);
    bytes memory result = asset.functionCall(assetData, "LOP: asset.call failed");
    if (result.length > 0) {
        require(abi.decode(result, (bool)), "LOP: asset.call bad result");
    }
}
```

Of particular concern here is that using assembly to both validate and modify this data in place is extremely susceptible to subtle errors that could very easily result in an arbitrary call to a caller-provided address – that is, to stealing of all user token approvals.

In addition to `LimitOrderProtocolRFQ`, both `UnoswapRouter` and `UnoswapV3Router` are written primarily in assembly.

Recommendation

Revisit the current usage of assembly. Consider the consequences of an assembly error in each place that it is used, the complexity and readability of the required assembly, and the gas savings provided.

As a concrete example, do not pass caller-provided data to arbitrary addresses, especially not data that has been modified multiple times by separate assembly blocks. For the 4 words of memory required for a `transferFrom()` call, the risk far outweighs the reward.

Avoid assembly blocks whenever possible. When necessary, make validation of the correctness of the assembly a high priority. Assembly blocks are easy to test in the happy path, but they have far more potential for unexpected behavior than high-level Solidity. This means that manual verification, fuzzing, symbolic execution, and other methods for ensuring correctness are well-warranted.

5.4 Unexpected ETH should be rejected Minor

Resolution

The development team has addressed this issue in commits `30f4067d9cac87280083407719dc4e436ed9ceab` and `c40696f8c8824f27bfd826dd65734d0a0c355225`. These changes have not been reviewed by the audit team.

Description

Many of the swap-related functions in the Aggregation Router are `payable`, as they give users the option to send ETH with the call which is then used in the swap. However, some of these methods do not reject calls with a nonzero `msg.value` in cases where ETH is not used in the swap.

Examples

The `uniswapV3SwapTo()` function below checks that the right amount of ETH was provided if the caller has asked that the callvalue be wrapped into WETH. However, it does not perform any additional checks on the callvalue in other cases.

code/contracts/UnoswapV3Router.sol:L50-L64

```
function uniswapV3SwapTo(
    address payable recepient,
    uint256 amount,
    uint256 minReturn,
    uint256[] calldata pools
) public payable returns(uint256 returnAmount) {
    uint256 len = pools.length;
    require(len > 0, "UNIV3R: empty pools");
    returnAmount = amount;
    bool wrapWeth = pools[0] & _WETH_WRAP_MASK > 0;
    bool unwrapWeth = pools[len - 1] & _WETH_UNWRAP_MASK > 0;
    if (wrapWeth) {
        require(msg.value == amount, "UNIV3R: wrong msg.value");
        _WETH.deposit{value: amount}();
    }
```

Recommendation

To prevent accidental locking of user funds, revert if the provided callvalue is nonzero on any path that does not utilize ETH. These checks should be added to `UnoswapV3Router.uniswapV3SwapTo()` and `ClipperRouter.clipperSwapTo()`

5.5 Malicious owner can use the AggregationExecutor to steal ETH funds Minor

Description

The `AggregationRouterV4` contract's `swap` function is the main point of interaction for users to trade in the system. Its first parameter takes a user-supplied `AggregationExecutor` interface (`caller`), which is called with user-supplied call data (`data`).

The given `AggregationExecutor` is then triggered using a low-level `call`. All gas from the current execution context will be forwarded to the external call. As the system does not have reentrancy guards, **any other system component can be reentered**.

code/contracts/AggregationRouterV4.sol:L121-L123

```
bytes memory callData = abi.encodePacked(caller.callBytes.selector, bytes12(0), msg.sender, data);
// solhint-disable-next-line avoid-low-level-calls
(bool success, bytes memory result) = address(caller).call{value: msg.value}(callData);
```

An attacker, e.g., manipulating the frontend or tricking users directly, can make users submit `swap` function calls with a `caller` address of their choosing.

code/contracts/AggregationRouterV4.sol:L88-L92

```
function swap(
    IAggregationExecutor caller,
    SwapDescription calldata desc,
    bytes calldata data
)
```

If the source token is ETH, an attacker can use a malicious contract with owner privileges to hijack the execution flow and reenter the `AggregationRouterV4` contract by calling the `destroy` function. Consequently, `selfdestruct` will be called, and the ETH funds in transit will be sent to the message sender.

code/contracts/AggregationRouterV4.sol:L155-L157

```
function destroy() external onlyOwner {
    selfdestruct(msg.sender);
}
```

The `swap` function's balance checks in the original execution frame will never be called as execution terminates immediately.

Recommendation

We recommend implementing reentrancy guards on administrative functions to prevent insider attacks and reduce the trust assumptions put into the owner role.

5.6 Opaque function signatures for `AggregationExecutor.callBytes()` Minor

Description

Currently, `AggregationExecutor` contracts are expected to expose a function `callBytes(address, bytes)`. The `AggregationRouter` takes user-supplied data and prepends the selector for `callBytes()` before calling to the `AggregationRouter`. Because the `AggregationRouter` holds user approvals, the development team wishes to avoid redeploying the router whenever possible.

To support multiple encodings for this bytes parameter without updates to the router, the `AggregationExecutor` contains a function with the desired types that is brute-forced to match the 4-byte function selector of `callBytes(address, bytes)`, which is `0x2636f7f8`. For example:

code/contracts/AggregationExecutor.sol:L56-L58

```
function func_0701rz5(address /* msgSender */, CallDescription[] calldata calls, bytes calldata /* approvedSenderAndSignature */) exte
    _makeCalls(calls);
}
```

This intentional use of function selector collisions introduces complexity for developers and users, making the encoded data that a user must sign indecipherable without access to the `AggregationExecutor` source code and could create unpredictable overlaps in argument encodings.

Recommendation

Implement the `callBytes(address, bytes)` function as written in `AggregationExecutor` contracts, and use Solidity's `abi.decode()` to decode the bytes parameter to the desired types. The development team currently avoids this due to gas concerns, but we assess that the risk and complexity introduced outweigh the benefits.

6 Recommendations

6.1 Improve inline documentation

Description

The source units hardly contain any inline documentation, making it hard to reason about methods and how they are supposed to be used. Consider adding natspec-format compliant inline code documentation, describe functions, what they are used for, and who is supposed to interact with them: document function or source-unit specific assumptions.

Not only will improved documentation increase the code's maintainability, but it also reduces the potential for future bugs. Furthermore, readability will be increased, allowing third-party auditors and new engineers to get started with the codebase quickly.

6.2 Deploy `AggregationRouterV4` from an EOA

Description

The Aggregation Router allows the contract owner to trigger a self-destruct at any time. If the contract were deployed via the `CREATE2` opcode using indeterminate initialization code, or if any contract within the deployment path were deployed using `CREATE2` *, the self-destructed router could be replaced with malicious code at the same address. This would allow all user token approvals to be stolen.

code/contracts/AggregationRouterV4.sol:L155-L157

```
function destroy() external onlyOwner {
    selfdestruct(msg.sender);
}
```

To minimize owner control of user assets, the Aggregation Router should be deployed in a transparently permanent manner. Ideally, this would mean deployment by an externally owned account (EOA).

* For example, contract X is deployed using `CREATE2` (not necessarily with indeterminate initialization code). Contract X deploys the Aggregation Router using `CREATE`. Both contract X and the Aggregation Router are self-destructed. This resets the nonce for both accounts. Because the destination address of `CREATE` relies on the deploying contract's nonce rather than the deployed contract's initialization code, the re-deployed Contract X can now deploy different contract code to the original Aggregation Router address.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
./AggregationRouterV4.sol	b54545e1f9ee517363fe3dfa2ab913f72555db45
./ClipperRouter.sol	6c6a877418e8f362bc0b2d04abefc470dfbfcb00
./helpers/ArgumentsDecoder.sol	00683488411bb10896080c25a82a8cc26f7f217b
./helpers/EthReceiver.sol	0792504da4f912f0ea49a2fc46b89046e2b547a0
./helpers/Permitable.sol	71f9d0e79b980cf70d80528163ad2ae763c5b050
./helpers/RevertReasonParser.sol	28714ae9c3a4011c9425c99621d51105df089c56
./helpers/UniERC20.sol	ca13eb469f49fb970dbddd2e87b1012466469378
./LimitOrderProtocolRFQ.sol	7b19d00eabf35a12c1fcdea4aaa28a22e01f710d
./UnoswapRouter.sol	72501583c4d9fbdc36672070fa633585bf742f54
./UnoswapV3Router.sol	f63d3b0a7c6ba6d1686bb4a359ffc471d5f77908

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.