arbitrary./execution

FULMIN LABS SECURITY ASSESSMENT

February 6, 2023

Prepared For:
Fulmin Labs
Prepared By:
Arbitrary Execution

Changelog:
January 20, 2023 *Initial report delivered*
February 6, 2023 *Final report delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's assessment of the Fulmin Labs Flood contracts. The Flood protocol aggregates decentralized exchanges and is designed to allow a user to swap tokens at an optimal price.

Three Arbitrary Execution engineers conducted this review over a 2-week period, from January 3, 2022 to January 17, 2022. The audited branch was `master` (commit hash `4df4ae532e8fddc774b484e06a5a5b98c2872261`) in the `fulmin-labs/flood-contracts` repository. The repository was private at the time of the engagement, so hyperlinks may not work for readers without access.

The team performed a manual review of the codebase with a focus on the interactions between different parties during the lifecycle of a trade. In addition to manual review, the team used Slither with AE's proprietary Slither detectors for automated static analysis.

The assessment resulted in findings ranging in severity from medium to note (informational). The medium finding discovered in the `AllKnowingOracle` could be used to effectively blacklist disputers, which are responsible for disputing a trade that has been filled at a non-optimal price. This could cause a user of the Flood protocol to submit trades that are undisputable and therefore may not be filled at the optimal price determined by the protocol.

FIX REVIEW UPDATE

Fulmin Labs has fixed or acknowledged all major issues identified in the engagement. The full breakdown of fixes can be found in the Fixes Summary section.

FIX REVIEW PROCESS

After receiving fixes for the findings shared with Fulmin Labs, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the *Update* section for a finding. For findings that the Fulmin Labs team chose not to address, the team's rationale is included in the update.

AUDIT OBJECTIVES

AE had the following high-level goals during this engagement:

- Identify smart contract vulnerabilities
- Evaluate adherence to Solidity best practices

SYSTEM OVERVIEW

SYSTEM COMPONENTS

The Flood protocol has three main components:

- Registry
- Oracle
- Book

REGISTRY

The registry stores information needed by the Flood protocol to function. It contains information such as the address of WETH (needed for the wrapping/unwrapping of the native currency of a network), accepted tokens for the protocol, and the address of the oracle to use when submitting or resolving a dispute. It also contains functions that can enable the use of ERC20 tokens with the protocol.

ORACLE

The `AllKnowingOracle` provides key functionality related to dispute resolution. It allows users to dispute filled trades and gives approved settlers the ability to settle a dispute.

BOOK

The book contract is used to submit a trade request, allow that trade request to be filled, dispute trades that are not filled at an optimal price, and cancel trade requests that haven't been filled yet.

There is currently one book that is used by the protocol, but it is likely that more will be deployed over time. What distinguishes books from each other are three key parameters. The first, dispute bond percentage, is a parameter that is used to determine how much of a bond is required in order to dispute a trade. The second parameter is the rebate given to the relayer in the event a trade has been disputed but has been resolved by a settler in the relayer's favor. The third key parameter is the refund a relayer receives when filling a trade. This refund is the core component of making a relayer's capital efficient which allows them to fill more trades than they would have been able to normally.

USER CATEGORIES

REQUESTERS

Users, also known as requesters, of the Flood protocol submit trades that contain three key parameters: the token they wish to swap, the token they wish to receive in return, and the minimum number of tokens they must receive. It is the responsibility of the user to set a reasonable minimum number of tokens.

RELAYERS

Relayers look for submitted trade requests in the `Book` contract. They are expected to listen off- chain for a `TradeRequested` event. If they decide that it is in their best interest to fill the order, they perform the fill and then the order will sit until a specified number of blocks have passed. Any party can then settle the trade and perform the token swap. Relayers receive a rebate when filling an order; this allows them to be more capital efficient and lets them fill more trades while they wait for a trade to be settled.

DISPUTERS

Disputers fulfill the role of ensuring that token swaps occur at optimal prices. Currently it is not known what oracle they will use to determine if a relayer is filling a trade at an optimal price. Disputers can dispute a trade that has been filled but is not settled. Once a trade has been disputed, the trade cannot proceed until a settler determines if the relayer or the disputer is correct. If the settler determines that the disputer was correct, then the disputer receives a reward. If instead the settler determines the relayer did use the optimal price, the disputer will lose tokens as a penalty.

SETTLERS

Settlers are approved via the `AllKnowingOracle` and are responsible for resolving disputed trades. This is another off-chain component of the protocol. Settlers listen for disputed trades via the `newRequest` or `TradeDisputed` event and respond accordingly by settling the dispute on the `AllKnowingOracle`.

PRIVILEGED ROLES

OWNER

The Registry contract uses the OpenZeppelin `Ownable2Step` contract. This grants the `owner` the ability to whitelist tokens and set the Oracle address that will be used in the Flood protocol. Ownership of the contract must be passed to another address in two steps. First, the current owner must set ownership to the new address. Then, the new address must claim the ownership in a second transaction.

The Oracle contract also uses the `Ownable2Step` access control mechanism. This allows the `owner` to approve addresses to act as settlers. Ownership can be transferred in the same way as the registry contract.

OBSERVATIONS

The Flood protocol has good documentation in the codebase which makes understanding the logic and intent easy. The repository has a robust Foundry test suite that exercises the core functionality of the protocol.

The Flood protocol was designed to fill trades at an optimal price. To accomplish this, the Flood protocol includes two safeguards: Traders are required to specify a minimum number of tokens that they expect to receive from a trade, and trades that use non-optimal token valuations can be disputed. Disputes can result in the user that filled the trade losing a non-trivial percentage of the tokens for which they traded.

The Flood protocol does not prevent a user from filling a trade and disputing that trade in the same block. Doing so effectively bypasses the dispute period enforced by `safeBlockThreshold` and prevents other protocol users from being able to dispute the trade. The result is an arbitrage opportunity, in which a user could receive more tokens by filling a trade using a non-optimal token price (but still meeting the required token minimum specified by the trader) and immediately disputing that trade in the same block. In such cases, the value the relayer would receive from the non-optimal trade, combined with the bond they receive from disputing that trade, would be greater than the value they would have received if they filled the trade at an optimal price.

Several key parts of the protocol implementation have not been finalized. The Flood protocol depends on settlers to function correctly, but the exact details of how settlers will operate, such as the frequency of dispute resolution and the means of determining the winning party, have not been finalized. The accounts that perform the duties of a settler are currently set by Fulmin Labs. This means that until a new process of approving settlers on chain is introduced, Fulmin Labs will act as the dispute resolution party.

Oracles are another part of the Flood protocol where the mechanism of operation is still to be determined. Fulmin Labs has stated they intend to initially use an oracle such as UMA to determine optimal prices for trades, but this is likely to change. Fulmin Labs has stated that they would like to use the technique found in the whitepaper "Optimal Routing for Constant Function Market Makers" to determine optimal prices in the future.

The aforementioned unknowns could require re-deployment of the Flood contracts, however the protocol does not use proxies. This can be problematic when attempting to add or change features of the protocol as there is no easy way to migrate storage from one contract to a newer version.

VULNERABILITY STATISTICS

Severity	Count
Critical	0
High	0
Medium	1
Low	3
Note	14

FIXES SUMMARY

Finding	Severity	Status
M01	Medium	Fixed in pull request #24
L01	Low	Fixed in pull request #37
L02	Low	Fixed in pull request #23
L03	Low	Fixed in pull request #22 and #42
N01	Note	Fixed in pull request #36
N02	Note	Fixed in pull request #34
N03	Note	Fixed in pull request #30
N04	Note	Fixed in pull request #31
N05	Note	Fixed in pull request #40
N06	Note	Fixed in pull request #32
N07	Note	Fixed in pull request #27
N08	Note	Fixed in pull request #36
N09	Note	Fixed in pull request #25
N10	Note	Fixed in pull request #26
N11	Note	Fixed in pull request #28
N12	Note	Fixed in pull request #29
N13	Note	Acknowledged
N14	Note	Fixed in pull request #33

FINDINGS

MEDIUM SEVERITY

[M01] REQUEST IDS ARE NOT UNIQUE

The `_getRequestId` function generates request IDs by creating a hash of packed data, which is constructed from the following sources of information:

- The address of the `Book` contract
- The address of the trade relayer
- The address of the trade disputer
- The address of the token used to pay the dispute bond
- The bond amount paid to dispute the trade

Because the potential values for each entry in this list are not unique to a specific trade, it is possible for a trader to create multiple trades whose disputes would generate identical request IDs. This is problematic because the protocol uses request IDs as indices into the `requests` mapping, the value of which is not zeroed after a dispute is resolved. The result is that all calls to `disputeTrade` that produce a duplicate request ID will `revert`.

To generate a request ID collision, the relayer, disputer, token, and bond amount values must all be identical to values used to produce a previous dispute. As a result, the impact of a request ID collision would be limited to a trader losing the protection of a disputer to ensure prices are being filled at optimal prices.

RECOMMENDATION

Consider adding a unique input to the set of values used to produce request IDs. This is already done in `_getTradeId` via the `tradeIndex` parameter, which is a counter that is incremented for each new trade.

UPDATE

Fixed in pull request #24 (commit hash `fe5a12caece014e77cf3afbc74ce0c72862f481c`), as recommended.

LOW SEVERITY

[L01] POTENTIALLY IMPRECISE TOKEN DISTRIBUTIONS

In the `Book` contract on line 253, line 302, and line 368, when computing the amount of tokens to send, the result of the integer division always rounds down.

This leads to the `Book` contract accumulating the remainder of the division. In the case of very small trades, the relayer can receive no tokens.

RECOMMENDATION

Consider calculating the amount of tokens left to pay out on a settled trade by subtracting the amount of tokens that have already been distributed from the total number of input tokens.

UPDATE

Fixed in pull request #37 (commit hash `a4738e754698170ab43283e0ed337346ec83e879`), as recommended.

[L02] SETTLERS CAN SETTLE THEIR OWN DISPUTES

Settlers can settle a dispute in which they are an involved party (either as the relayer or the disputer). Since settlers determine the outcome of a dispute, they could use that authority to resolve disputes in which they are a party to in their own favor. The settler role is granted to addresses at the discretion of Fulmin Labs.

RECOMMENDATION

Consider adding a check to `settle` that ensures that `msg.sender` (the settler) is not equal to `request.proposer` nor `request.disputer`.

UPDATE

Fixed in pull request #23 (commit hash `0aec25e83229860b0b1c29218d536c8ce70a333a`), as recommended.

[L03] USE OF `TRANSFER` FUNCTION TO SEND ETHER

On line 398 and line 407 of the `Book` contract, `transfer` is used to send ether to a recipient address.

The use of the `transfer` function to send ether is no longer recommended. The `transfer` function forwards a fixed (2300) amount of gas, and compatibility with receiving contracts may be broken if opcode gas costs are altered in the future.

RECOMMENDATION

Consider using the low level `call` function to transfer ether to an address.

UPDATE

Fixed in pull request #22 (commit hash `7312b69d4c1dc595cfb740c35a5be804901a8abc` and pull request #42(commit hash `b1d587b6b179ef903c7c74eb66579e50046a41e7`).

NOTE SEVERITY

[N01] `_DELETETRADE` DOES NOT DELETE ALL TRADE VARIABLES

In the `Book` contract, the `_deleteTrade` function deletes the entries in the `filledAtBlock`, `filledBy`, and `status` mappings, but not the entries in the `unwrapOutput` or `isEthTrade` mappings.

RECOMMENDATION

Delete all storage variables related to a trade when executing the `_deleteTrade` function.

UPDATE

Fixed in pull request #36 (commit hash `e5a31a94a10cd21f9a0b55a54e0c3ad4842b56c3`), as recommended.

[N02] USE OF FLOATING COMPILER VERSION PRAGMA

All of the contracts in this repository float their Solidity compiler versions (e.g. `pragma solidity ^0.8.17`).

Locking the compiler version prevents accidentally deploying the contracts with a different version than what was used for testing. The current pragma prevents contracts from being deployed with an outdated compiler version, but still allows contracts to be deployed with newer compiler versions that may have higher risks of undiscovered bugs.

It is best practice to deploy contracts with the same compiler version that is used during testing and development.

RECOMMENDATION

Consider locking the compiler pragma to the specific version of the Solidity compiler used during testing and development.

UPDATE

Fixed in pull request #34 (commit hash `5f5a3707eb6c93924babcec1f636fa6700e9cf72`), as recommended.

[N03] INACCURATE DOCUMENTATION

The following comments are inaccurate, either in their content or placement in the code:

- Line 94: The `getRequestId` docstring refers to a `BookSingleChain` contract that does not exist in the repository
- Line 133: The `ask` docstring refers to a `BookSingleChain` contract that does not exist in the repository
- Line 169: The `settle` docstring states that the function calls back into the `proposer`, but instead calls back into the `requester`
- Line 61: The `TradeStatus` comment is not placed directly above the `enum` entry it describes
- Line 133: The `requestTrade` docstring refers to a `feeCombination` structure that does not exist in the repository
- Line 183: The `cancelTrade` docstring describes the role of the function as filling trades, when function is responsible for cancelling trades
- Line 217: The `fillTrade` docstring states the function is called internally, but is `external` type

RECOMMENDATION

Consider updating the comments listed above to accurately describe the code to which they refer.

UPDATE

Fixed in pull request #30 (commit hash `5f78776df5785422092b3e59a207d3fd5149b179`), as recommended.

[N04] LACK OF NATSPEC DOCUMENTATION

The following functions within the codebase either lack docstrings or have incomplete NatSpec documentation, such as omitting the `@title`, `@param`, or `@return` comments:

- Line 80: The `whitelistSettler` function
- Line 119: The `ask` function (lacks `@return`)
- Line 230: The `fillTrade` function (lacks `@param` for `data`)
- Line 469: The `_getTradeId` function (lacks `@param` for `trader`)
- Line 184: The `receive` function

Lack of complete documentation makes understanding and interacting with the codebase more difficult.

RECOMMENDATION

The Solidity documentation recommends NatSpec for all public interfaces (everything in the ABI). Consider implementing NatSpec-compliant docstrings for all public and external functions.

A good example is the OpenZeppelin ERC20 contract. It follows the NatSpec guidelines, and provides contract documentation that gives additional information about context and usage.

UPDATE

Fixed in pull request #31 (commit hash `11a21860622c9c84822fc1eb3ab69d60129a8bd8`), as recommended.

[N05] UNNECESSARY INTERNAL HELPER FUNCTION

In the `AllKnowingOracle` contract, the `getRequestId` function is declared `external` but passes all of its parameters to the `internal` helper function `_getRequestId`.

The contract code size can be reduced by declaring `getRequestId` public and inlining the function body of `_getRequestId`.

RECOMMENDATION

Consider inlining the `_getRequestId` function into the `getRequestId` function, and setting the function visibility to `public`.

UPDATE

Fixed in pull request #40 (commit hash `1f7895ac6a726b67734ba13809b86d82c8fcc913`), as recommended.

[N06] NO SANITY CHECKS ON CONSTRUCTOR PARAMETERS

In the `Book` contract, the `constructor` function lacks sanity checks on the values of the `_registry` and `_safeBlockThreshold` parameters. Since these variables are immutable, the only corrective measure available if an incorrect value is used is re-deployment.

RECOMMENDATION

Consider adding sanity checks to the `_registry` and `_safeBlackThreshold` constructor parameters to ensure the parameters are not zero.

UPDATE

Fixed in pull request #32 (commit hash `4bb38fe330105739dfb8dbf5f6650c77f101c6d4`), as recommended.

[N07] INCOMPLETE STATE CHECK IN `SETTLE`

On line 153 of the `settle` function, the state of a request is only checked to ensure that it has not already been settled. The only valid state in which a request can be settled is `Pending`. This means that a request with a state of `Uninitialized` will pass the check but will revert later in the function as it attempts to transfer ERC20 tokens.

RECOMMENDATION

Consider checking the necessary precondition explicitly.

UPDATE

Fixed in pull request #27 (commit hash `2cb79a1854c831fbe844b47f522bb39406904cef`), as recommended.

[N08] LACK OF ENCAPSULATION OF TRADE DATA

In the `Book` contract, the following storage variables use a mapping to store the properties of a requested trade:

- `filledAtBlock`
- `filledBy`
- `status`
- `unwrapOutput`
- `isEthTrade`

Using a struct that contains the details of a trade can make the code easier to understand, and can help ensure that all variables related to a trade are properly cleaned up when the `cancelTrade` function is executed.

RECOMMENDATION

Consider refactoring the code to eliminate the multiple mappings for these variables, in favor of a single mapping that stores a new struct type that contains all these related trade variables. An example implementation can be seen below:

```
struct TradeData {
    // Maps each trade id to the block it was filled at. A value of 0 means
it was not filled yet.
    mapping(bytes32 => uint256) public filledAtBlock,
    // A mapping from a trade id to the relayer filling it.
    mapping(bytes32 => address) public filledBy,
    // A mapping from trade id to an enum representing the state of the
trade.
    mapping(bytes32 => TradeStatus) public status,
    // A mapping from a trade id to if the recipient wants to unwrap their
output token.
    mapping(bytes32 => bool) public unwrapOutput,
    // A mapping from a trade id to whether the trade was requested with ETH.
    mapping(bytes32 => bool) public isEthTrade
}

mapping(bytes32 => TradeData) trades;
```

UPDATE

Fixed in pull request #36 (commit hash `e5a31a94a10cd21f9a0b55a54e0c3ad4842b56c3`), as recommended.

[N09] UNNECESSARY TYPE CAST

On line 176 of `Book.sol`, the state variable `weth` is unnecessarily cast to its declared type of `IWETH9`.

RECOMMENDATION

Consider removing the unnecessary type cast.

UPDATE

Fixed in pull request #25 (commit hash `4067d6ae816b954648b662e64a238993877630a6`), as recommended.

[N10] UNNECESSARY CALL TO SAFEAPPROVE

On lines 351-358 of the `Book` contract, the oracle is given a token allowance of twice the bond amount using `safeApprove`. It then performs an external message call to the `ask` function in the oracle which transfers the tokens to the oracle. The allowance is then set to zero using `safeApprove`. The `ask` function in the oracle will transfer the full allowance set in the first call to `safeApprove`, so the second call to `safeApprove` is unnecessary because the allowance remaining will already be set to zero.

RECOMMENDATION

Consider removing the second call to `safeApprove`.

UPDATE

Fixed in pull request #26 (commit hash `4d154ed23e47a9ab10699817c2a15b04487877c9`), as recommended.

[N11] UNNECESSARY FUNCTION PARAMETER

In the `Book` contract, the `cancel` function checks that the `trader` function parameter must be equal to `msg.sender`, or the transaction will revert. As a consequence, `msg.sender` could be used in place of the `trader` parameter, and the `trader` parameter can be removed from the function definition.

RECOMMENDATION

Consider removing the unnecessary `trader` parameter.

UPDATE

Fixed in pull request #28 (commit hash `1c661ea3e0e055a47c3811d144cdcaf01994369d`), as recommended.

[N12] UNNECESSARY CONDITION IN `IF` STATEMENT

On line 151 of `Book.sol`, the condition asserts that `msg.value` equals `amountIn`. This check is unnecessarily repeated on line 174.

RECOMMENDATION

Consider removing the unnecessary `if` condition.

UPDATE

Fixed in pull request #29 (commit hash `90041cb3785feac9840b390e1dff5b79b33345d3`), as recommended.

[N13] UNUSED STORAGE VARIABLES

The following storage variables are defined but not used:

- Line 61: The `registry` variable is defined but not used.
- Line 78: The `feePct` variable is defined but not used.

RECOMMENDATION

Consider removing all unused variables.

UPDATE

Acknowledged. Fulmin Labs' statement for this issue:

> *… while not used in the contract logic itself, those variables are really helpful for relayers & disputers (especially the `feePct`), the easiest way to agree on which `feePct` is being used in a Book is to look it up onchain. Otherwise we would need some other of trustless "data store" to put the `feePct` in, and would need to map it to each book.*

[N14] TOKEN WHITELISTING FUNCTION DOES NOT CHECK IF ADDRESS IS A CONTRACT

The `_whitelistToken` function does not check whether the supplied address points to a contract. As a result, the contract owner can whitelist an EOA as a token contract.

RECOMMENDATION

Consider adding a check to the `_whitelistToken` function that ensures the provided address points to a contract. See OpenZeppelin's `isContract` function implementation for an example of this check.

UPDATE

Fixed in pull request #33 (commit hash `150ceae6bfe45d6d4ad3a9fc4377ac5fc490a14f`). The check was added to both of the functions that call `_whitelistToken`, as opposed to `_whitelistToken` itself.

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	Definition
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
src/AllKnowingOracle.sol
src/Book.sol
src/FloodRegistry.sol
src/interfaces/IFloodFillCallback.sol
src/interfaces/IWETH9.sol
```