Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

catalyst

Cross Cats

Veridise Inc.

December 3, 2024

► **Prepared For:**

Cata Labs
https://catalabs.org/

► **Prepared By:**

Jon Stephens
Benjamin Sepanski
Nicholas Brown

► **Contact Us:**

contact@veridise.com

► **Version History:**

Dec. 3, 2024	V1
Oct. 28, 2024	Initial Draft

Contents

1 Executive Summary

From Oct. 14, 2024 to Oct. 25, 2024, Cata Labs engaged Veridise to conduct a security assessment of their Cross Cats. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `354bb0b21`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The security assessment covered a protocol that allows users (referred to as swappers) to place orders for token swaps on the same chain or across chains. These swaps can be filled by other users of the protocol (called fillers). The two main components of Cross Cats are reactors and oracles. The reactors are used to keep track an order's state, and the oracles are used to prove whether an order has been filled.

More specifically, reactors are used to track a cross-chain order, where an order can be in one of several states. An order starts in an "unfilled" state in which the user has specified the inputs and outputs to the swap along with information about how to confirm that an order is filled. This order can also contain "reactor-specific" information if a given reactor provides custom trading behavior. The "unfilled" order is communicated to the fillers who must submit the order along with a PERMIT2 signature from the user to "initiate" the order, effectively claiming the order on behalf of the filler. This also collects the inputs to the swap from the swapper, which the reactor will escrow, and will collect collateral from the filler so that they are incentivized to fill the order. Once the order is claimed, a filler should fill the order with the help of the remote oracle (on chains other than bitcoin). At this point, they can use a local oracle to prove the order has been filled and reclaim their collateral along with the inputs to the swap. Alternatively, they can wait until the "challenge deadline" has passed and claim the funds without providing proof. At any time before the challenge deadline has been exceeded, any user can issue a challenge to an order by providing challenge collateral. At this time the filler must prove that the order has been filled or, if no proof is provided before the deadline, have the order be marked as fraud. In the case of fraud the inputs are provided back to the swapper and filler's collateral is split between the swapper and challenger.

Oracles work to support the above reactor by proving that an order has been filled. In the case of evm-based chains, these oracles facilitate the fulfillment process and can communicate which swap outputs have been filled to other cross-chain oracles. In the case of bitcoin, the oracle uses an on-chain bitcoin light-client to prove that the specified payment has been made. Information about filled "outputs" will then be provided to the reactor upon request.

Code Assessment. The Cross Cats developers provided the source code of the Cross Cats contracts for the code review. The source code appears to be original code written by the Cross Cats developers. It contains documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts understanding of the code, the Cross Cats developers met with the Veridise security analysts to explain the protocol at a high level and provide additional documentation.

The source code contained a test suite, which the Veridise security analysts noted tested the project components with some success and failure conditions.

Summary of Issues Detected. The security assessment uncovered 17 issues, 1 of which is assessed to be of high or critical severity by the Veridise analysts. Specifically, an integer overflow in the `GeneralisedIncentivesOracle` can result in arbitrary outputs being able to be proven (V-CRC-VUL-001). The Veridise analysts also identified 3 medium-severity issues, including several frontrunning risks (V-CRC-VUL-004, V-CRC-VUL-002) and the ability for an output to fill multiple orders on different chains (V-CRC-VUL-003) as well as 6 low-severity issues, 5 warnings, and 2 informational findings. The Cross Cats developers have either fixed or responded to all of the issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Cross Cats:

1. This project places heavy trust in the content of the user orders with minimal protections for users who have signed maliciously constructed orders or might not have understood the impact of their decisions. The Veridise security analysts would emphasize the importance of documenting the risks associated with poorly constructed orders and providing a set of validations that the developers would recommend users perform.
2. The security analysts also noted that there is the potential for collusion between different parties involved in a swap. Such collusion could be difficult to identify and likely would obfuscate the honesty of individual actors. The Veridise security analysts would also recommend providing information to users about how to identify collusion and the potential impacts of colluding entities.
3. Finally, a significant amount of trust is placed in the oracle by both the filler and swapper and therefore oracles must be rigorously evaluated. It is unclear, however, whether all users will be capable of performing this evaluation. We would therefore recommend providing resources to help users with this evaluation, including providing a best-practices for oracles and maintaining a list of oracles that have misbehaved in the past.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Cross Cats	`354bb0b21`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 14–Oct. 25, 2024	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	0	0	0
Medium-Severity Issues	3	1	0
Low-Severity Issues	6	5	4
Warning-Severity Issues	5	4	3
Informational-Severity Issues	2	2	2
TOTAL	17	13	10

Table 2.4: Category Breakdown.

Name	Number
Data Validation	4
Frontrunning	3
Logic Error	3
Reentrancy	2
Integer Overflow	1
Usability Issue	1
Authorization	1
Collusion	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of the Cross Cats smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Can a filler prove that they have filled an order without ever filling it?
- Can a swapper prevent an order that has been filled from being proven to be filled?
- Can frontrunning be used to take advantage of users of the protocol?
- Are inputs validated correctly?
- Can fillers hide malicious behavior from the rest of the protocol?
- Is it possible for different parties to collude and what advantages can this collusion bring?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this security assessment was limited to the `src/` folder of the source code provided by the Cross Cats developers, which contains the smart contract implementation of the Cross Cats.

Methodology. Veridise security analysts inspected the provided tests and read the Cross Cats documentation. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise security analysts regularly met with the Cross Cats developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

The likelihood of a vulnerability is evaluated according to the Table 3.2.

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-CRC-VUL-001	Overflow Allows Arbitrary Outputs to be . . .	Critical	Fixed
V-CRC-VUL-002	Filler can Frontrun Initiate	Medium	Acknowledged
V-CRC-VUL-003	Output can fill Multiple Orders on . . .	Medium	Won't Fix
V-CRC-VUL-004	Order Fulfillment Proof can be Frontrun	Medium	Won't Fix
V-CRC-VUL-005	Few user Protections Performed	Low	Won't Fix
V-CRC-VUL-006	Missing Checks that Tokens are Contracts	Low	Fixed
V-CRC-VUL-007	Governance Fees can change after Order . . .	Low	Fixed
V-CRC-VUL-008	Fillers can Impact Reputation of Other Fillers	Low	Fixed
V-CRC-VUL-009	Multiple Collusion Opportunities	Low	Acknowledged
V-CRC-VUL-010	Bitcoin Token with Unknown . . .	Low	Fixed
V-CRC-VUL-011	Order Dispute can be Frontrun	Warning	Acknowledged
V-CRC-VUL-012	Read-Only Reentrancy in . . .	Warning	Won't Fix
V-CRC-VUL-013	Filler can Lose Funds when Purchasing Order	Warning	Fixed
V-CRC-VUL-014	Parsed BitcoinAddress may be Invalid	Warning	Fixed
V-CRC-VUL-015	One Chain ID may map to Multiple Escrows	Warning	Fixed
V-CRC-VUL-016	Function Missing Override	Info	Fixed
V-CRC-VUL-017	Protocol Can Introduce Reentrancies into . . .	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-CRC-VUL-001: Overflow Allows Arbitrary Outputs to be Proven

Severity	Critical	Commit	354bb0b
Type	Integer Overflow	Status	Fixed
File(s)	GeneralisedIncentivesOracle.sol		
Location(s)	_encode, _decode		
Confirmed Fix At	92112af		

To prove that cross-chain orders have been filled, the cross-cat project relies on cross-chain oracles to facilitate order output fulfillment and communicate information about filled outputs. To do so, an `OutputDescription` along with its associate fill deadline must be communicated via a cross-chain bridge. Cross-cats uses the `_encode` function shown below to convert an array of `OutputDescription`, `fillDeadline` pairs to bytes for communication. During the encoding process, however, an oracle will cast the length of the `OutputDescription.remoteCall` to 2 bytes without checking the actual number of bytes in the string as shown below.

```
function _encode(
    OutputDescription[] calldata outputs,
    uint32[] calldata fillDeadlines
) internal pure returns (bytes memory encodedPayload) {
    uint256 numOutputs = outputs.length;
    encodedPayload = bytes.concat(bytes1(0x00), bytes2(uint16(numOutputs)));
    unchecked {
        for (uint256 i; i < numOutputs; ++i) {
            OutputDescription calldata output = outputs[i];
            // if fillDeadlines.length < outputs.length then fillDeadlines[i] will
    fail with out of index.
            uint32 fillDeadline = fillDeadlines[i];
            encodedPayload = bytes.concat(
                ...
                bytes2(uint16(output.remoteCall.length)),
                output.remoteCall
            );
        }
    }
}
```

Snippet 4.1: Definition of the `_encode` function

Doing so can cause the length of the remote call to overflow which will cause the given `OutputDescription` to be improperly decoded using the `_decode` function shown below. This is because the remote call length is used to determine the number of bytes in `remoteCall` and, importantly, the number of bytes to the next output that was sent. In a case where the `remoteCall` length overflows, the next output will be read from within the `remoteCall` bytes rather than after them. This can cause `_decode` to return `OutputDescription`, `fillDeadline` pairs that were not proven by the remote oracle.

```
function _decode(
    bytes calldata encodedPayload,
    bytes32 remoteOracle
```

```
) internal pure returns (OutputDescription[] memory outputs, uint32[] memory
    fillDeadlines) {
    unchecked {
        uint256 numOutputs = uint256(uint16(bytes2(encodedPayload[NUM_OUTPUTS_START:
    NUM_OUTPUTS_END])));

        outputs = new OutputDescription[](numOutputs);
        fillDeadlines = new uint32[](numOutputs);
        uint256 pointer = OUTPUT_TOKEN_START;
        for (uint256 outputIndex; outputIndex < numOutputs; ++outputIndex) {
            ...

            uint256 remoteCallLength = uint16(
                bytes2(encodedPayload[pointer:pointer += (REMOTE_CALL_LENGTH_END -
    REMOTE_CALL_LENGTH_START)])
            );

            outputs[outputIndex] = OutputDescription({
                remoteOracle: remoteOracle,
                token: token,
                amount: amount,
                recipient: recipient,
                chainId: chainId,
                remoteCall: encodedPayload[pointer:pointer += remoteCallLength]
            });
        }
    }
}
```

Snippet 4.2: Definition of the `_decode` function

Impact Since the output of `_decode` is used to set `OutputDescription`, `fillDeadline` pairs as proven, this function could allow incorrect outputs to be marked as proven on the source oracle. Additionally, it should be noted that an attacker can control the contents of `remoteCall`, which would allow an attacker to arbitrarily set outputs as proven in the source chain's oracle by providing a large `OutputDescription`.

Recommendation Validate that the length of `remoteCall` fits into 2 bytes.

Developer Response The developers have implemented the recommendation.

4.1.2 V-CRC-VUL-002: Filler can Frontrun Initiate

Severity	Medium	**Commit**	354bb0b
Type	Frontrunning	**Status**	Acknowledged
File(s)	`BaseReactor.sol`		
Location(s)	Initiate		
Confirmed Fix At	N/A		

The cross-cats project allows users to submit signed orders which fillers will "claim" by invoking the initiate function. Once an order has been claimed, that filler will be responsible for filling the order and will receive the inputs to the cross-chain swap once filled. Notably, the first person to invoke `initiate` will "claim" it, allowing fillers to frontrun others during the claiming process. Notably, this may not be in the best-interest of users as fillers can frontrun claims of orders that are advantageous to them. Note, this could be an order that was created in a way that could benefit the filler (see V-CRC-VUL-005) or orders that specify oracles controlled by the filler, potentially allowing filler/oracle collusion.

Impact Without a method of whitelisting or blacklisting fillers, it will be difficult for users to avoid bad-acting or colluding fillers.

Recommendation The `LimitOrderReactor` in general and `DutchOrderReactor` after the `slopeStartingTime` do not restrict order fillers. Consider allowing users to define filler whitelists in both reactor types.

Developer Response The developers have acknowledged this issue.

4.1.3 V-CRC-VUL-003: Output can fill Multiple Orders on Different Chains

Severity	Medium	**Commit**	354bb0b
Type	Logic Error	**Status**	Won't Fix
File(s)	`BaseOracle.sol`		
Location(s)	_isProven()		
Confirmed Fix At	N/A		

In this protocol, Oracles record whether an output has been proven to be filled by indexing using a hash of the `OutputDescription` and by the `fillDeadline`. This means that multiple orders with identical outputs may be proven to be filled even if only one of them was filled. The developers indicate that protections are in place in the frontend to reduce the risk of collision, but collisions can happen on-chain in a variety of ways which can be used as part of an effective social engineering attack on a user.

```
1    function _isProven(OutputDescription calldata output, uint32 fillDeadline)
     internal view returns (bool proven) {
2        bytes32 outputHash = _outputHash(output);
3        return _provenOutput[outputHash][fillDeadline];
4    }
```

Snippet 4.3: Definition of `_isProven()`

Impact One example of a non-obvious attack is the following:

The `GeneralisedIncentivesOracle.submit()` function will send a cross chain message to any destination that a specific output has been paid out. This means that orders with identical outputs could be submitted on two different source chains and they could both be proven as filled when only one of them was filled by calling the `submit()` function twice to send to both chains. This would also bypass the recommended checks to prevent collisions in V-CRC-VUL-005 because the reactor and local oracle for each order would have no knowledge of the other order.

Recommendation EVM oracles should include the order nonce, source chain, and settlement contract as part of the index for the proof to reduce the risk of collisions on these chains. This doesn't solve the problem with collisions involving Bitcoin outputs, but it reduces the attack surface for the protocol.

Developer Response The developers have indicated that they do not plan to fix this issue.

4.1.4 V-CRC-VUL-004: Order Fulfillment Proof can be Frontrun

Severity	Medium	**Commit**	354bb0b
Type	Frontrunning	**Status**	Won't Fix
File(s)	`BaseReactor.sol`		
Location(s)	proveOrderFulfilment		
Confirmed Fix At	N/A		

The cross-cats protocol defines several deadlines that control its behavior. One such deadline is the `challengeDeadline`, before which the filler of an order cannot receive payment unless they provide proof via the `proveOrderFulfilment` function. Another is the `orderPurchaseDeadline`, which allows a filler to specify a period during which the order may be purchased from them using the `purchaseOrder` function. Importantly, both `proveOrderFulfillment` and `purchaseOrder` can be called when an order is claimed or challenged, potentially resulting in unexpected consequences for the filler. This is due to the fact that if an order is filled before the `orderPurchaseDeadline` expires, another user can purchase the order before `proveOrderFulfilment` is called—either by frontrunning the transaction or when evidence is posted on the destination chain.

Impact Since it appears that orders are often likely to be offered for purchase at a discount, doing so would allow a user to safely collect the discount (minus the governance fee) without providing any service.

Recommendation While the protocol cannot perfectly protect fillers against this risk, we would recommend:

1. Informing fillers that this is a risk so that they adjust `orderPurchaseDeadline` as necessary.
2. Add a check to `purchaseOrder` to determine whether the order has already been filled.

Developer Response The developers have indicated that they do not plan to fix this issue.

4.1.5 V-CRC-VUL-005: Few user Protections Performed

Severity	Low	**Commit**	354bb0b
Type	Data Validation	**Status**	Won't Fix
File(s)	`BaseReactor.sol`		
Location(s)	initiate()		
Confirmed Fix At	N/A		

When a user makes an order, they will generate a signature for the details of that order. When a filler decides to fill a user's order, they will call `initiate()` with the user's order and signature. The `initiate()` function places heavy trust on the contents of the order signed by the user which can cause problems if users make mistakes.

```
1    // Check if this is the right contract.
2    if (order.settlementContract != address(this)) revert InvalidSettlementAddress();
3    // Check if the expected chain of the order is the rigtht one.
4    if (order.originChainId != block.chainid) revert WrongChain(uint32(block.chainid)
     , order.originChainId);
5
6    // Check if initiate Deadline has passed.
7    if (order.initiateDeadline < block.timestamp) revert InitiateDeadlinePassed();
8
9    // The order initiation must be less than the fill deadline. Both of them cannot
     be the current time as well.
10   // Fill deadline must be after initiate deadline. Otherwise the solver is prone
     to making a mistake.
11   if (order.fillDeadline < order.initiateDeadline) revert InitiateDeadlineAfterFill
     ();
```

Snippet 4.4: The checks performed in `initiate()`

These are some notable checks that are not included:

- A check that the filler is required to provide a reasonable amount of collateral.
- A check that the collateral required for a challenger has a similar magnitude to the collateral required by the filler.
- A check that a user doesn't have multiple orders with identical outputs.

Impact This issue is marked as low severity because the developers indicated that the details of an order should be trusted, and users are expected to use a frontend that will perform additional checks. However, it is still possible for bugs in the frontend or social engineering attacks to lead to users signing harmful orders.

Here are some examples of issues that can arise if a user signs an order with an unfavorable configuration:

- A malicious filler can DoS orders that require little or no collateral by calling `initiate()` and never filling the order. This can become even worse if the collateral required for the challenger is high enough that challengers aren't incentivized to challenge the order. In such a situation the malicious filler can steal the inputs of the order if the order never gets challenged.

- If the challenger collateral is zero or low, malicious challengers may challenge the order regardless of whether it was filled which can result in unnecessary work by the protocol. Low challenger collateral will also make issues like V-CRC-VUL-011 worse.
- If a user is tricked into signing multiple Bitcoin orders with the same outputs (or any type of order with the same outputs and fill deadline), a filler can initiate all such orders and receive the inputs while only filling one of the orders. See V-CRC-VUL-003 for details on an attack like this.

Recommendation

- Add a check that the filler collateral is non-zero or at least a certain percentage of the order inputs
- Add a check that the challenger collateral has a comparable magnitude to the filler collateral (not significantly larger and not small enough to allow for frivolous challenges)
- Add a check to reduce the risk of a user signing an order with identical outputs
  - None of these will completely solve the problem but can make a social engineering attack more difficult:
    - A check that the order isn't already proven to be filled when initiating an order (to ensure that an order with identical outputs hasn't already occurred in a different reactor)
    - A check that the order outputs hash hasn't already been used in the current reactor

Developer Response The developers have indicated that they do not plan to fix this issue.

4.1.6 V-CRC-VUL-006: Missing Checks that Tokens are Contracts

Severity	Low	**Commit**	354bb0b
Type	Data Validation	**Status**	Fixed
File(s)	`ReactorPayments.sol`, `BaseReactor.sol`		
Location(s)	_deliverTokens()		
Confirmed Fix At	`b75a18a`		

`_deliverTokens()` is used to pay the input tokens to either the filler or back to the user depending on the status of the order. This function will iterate over the tokens and use `safeTransfer()` to transfer the specified token to the recipient. This `safeTransfer()` will succeed even if the token does not exist.

```
1    function _deliverTokens(Input[] calldata inputs, address to, uint256 fee)
     internal virtual {
2        // Read governance fee.
3        uint256 numInputs = inputs.length;
4        for (uint256 i; i < numInputs; ++i) {
5            Input calldata input = inputs[i];
6            address token = input.token;
7            // Collect governance fee. Importantly, this also sets the value as
     collected.
8            uint256 amount = input.amount;
9            amount = fee == 0 ? amount : _collectGovernanceFee(token, amount, fee);
10           // We don't need to check if token is deployed since we
11           // got here. Reverting here would also freeze collateral.
12           SafeTransferLib.safeTransfer(token, to, amount);
13       }
14   }
```

Snippet 4.5: Snippet from `_deliverTokens()`

Impact If a user specifies a token that doesn't exist, the filler will not receive the full inputs that they will be expecting. Since permit2 will also succeed even if the token doesn't exist, an order could make it to this point with nonexistent token inputs. This makes it easier for fillers to make mistakes.

Recommendation There should be a check in `BaseReactor.initiate` that the input tokens exist.

Developer Response The developers implemented this check in `Permit2Lib.toPermit` which is called by `BaseReactor.initiate`.

4.1.7 V-CRC-VUL-007: Governance Fees can change after Order Acceptance

Severity	Low	**Commit**	354bb0b
Type	Usability Issue	**Status**	Fixed
File(s)	`CanCollectGovernanceFee.sol`		
Location(s)	setGovernanceFee()		
Confirmed Fix At	`d80bc85`		

A governance fee is collected from the order inputs when sending funds to the filler after an order has been completed successfully. This governance fee amount is managed by the `owner` of the reactor by calling `setGovernanceFee` . Since changes to the governance fee are applied to existing orders as well as new orders, an increase to the governance fee could cause fillers to lose money on orders with low margins that they had initiated or purchased previously.

```
1    /**
2     * @notice Sets a new governanceFee. Is immediately applied to orders initiated
     after this call.
3     * @param newGovernanceFee New governance fee. Is bounded by MAX_GOVERNANCE_FEE.
4     */
5    function setGovernanceFee(
6        uint256 newGovernanceFee
7    ) external onlyOwner {
8        if (newGovernanceFee > MAX_GOVERNANCE_FEE) revert GovernanceFeeTooHigh();
9        uint256 oldGovernanceFee = governanceFee;
10       governanceFee = newGovernanceFee;
11
12       emit GovernanceFeeChanged(oldGovernanceFee, newGovernanceFee);
13   }
```

Snippet 4.6: Definition of `setGovernanceFee()`

Impact Fillers with a high volume of active orders may lose significant amounts of money if the governance fee is increased too much.

Recommendation The owner of the contract should be a timelock that allows enough time for active orders to be resolved before the governance fee change is executed. Alternatively the governance fee could be attached to the `OrderContext` so changes to the governance fee would only impact future orders.

Developer Response The developers have implemented a timelock solution for changing the governance fee.

4.1.8 V-CRC-VUL-008: Fillers can Impact Reputation of Other Fillers

Severity	Low	**Commit**	354bb0b
Type	Authorization	**Status**	Fixed
File(s)	`BaseReactor.sol`		
Location(s)	modifyOrderFillerdata, purchaseOrder, Initiate		
Confirmed Fix At	`cb86b8b`		

We have found that in applications similar to cross-cats, it is important for users to understand the reputation of entities they interact with so that they can understand the trust-worthiness of oracles and fillers. The cross-cats protocol, however, allows fillers to manipulate the reputation of other fillers. As an example, consider the `modifyOrderFillerdata` function. This function can be used by a filler to manipulate the `OrderContext` associated with their orders including after the order has been filled or proven to be a fraud. This would allow a filler to manipulate the identity of a filler associated with a fraudulent order which would be difficult to determine without inspecting the contract's logs.

```
1  function modifyOrderFillerdata(OrderKey calldata orderKey, bytes calldata fillerData)
       external {
2      bytes32 orderKeyHash = _orderKeyHash(orderKey);
3      OrderContext storage orderContext = _orders[orderKeyHash];
4
5      address currentFiller = orderContext.fillerAddress;
6
7      // This line also disallows modifying non-claimed orders.
8      if (currentFiller == address(0) || currentFiller != msg.sender) revert OnlyFiller
       ();
9
10     // Decode filler data.
11     (
12         address newFillerAddress,
13         uint32 newOrderPurchaseDeadline,
14         uint16 newOrderPurchaseDiscount,
15         bytes32 newIdentifier,
16     ) = FillerDataLib.decode(fillerData);
17
18     // Set new storage.
19     if (newFillerAddress != currentFiller) orderContext.fillerAddress =
       newFillerAddress;
20     orderContext.orderPurchaseDeadline = newOrderPurchaseDeadline;
21     orderContext.orderPurchaseDiscount = newOrderPurchaseDiscount;
22     orderContext.identifier = newIdentifier;
23
24     emit OrderPurchaseDetailsModified(orderKeyHash, fillerData);
25 }
```

Snippet 4.7: Definition of the `modifyOrderFillerdata` function

Impact If a filler frequently fails to fill orders they can make it appear as though they have filled orders perfectly and can pass the blame onto an honest filler.

Recommendation In addition to `modifyOrderFillerdata`, a malicious filler can similarly affect another's reputation by purchasing (purchase funds would be sent back to malicious filler) or claiming an order on another filler's behalf (presumably for small orders or orders with small collateral). To prevent this, we would recommend the following:

1. Allow fillers to whitelist who may claim or purchase an order for them.
2. Disallow changes to the order context once the order is in a final state (Proven, OptimisticallyFilled, Fraud).
3. Consider providing a 2-step transfer process where a new filler must accept the order when transferred using `modifyOrderFillerdata`.

Developer Response The developers have disallowed changes to the order context when the order is in a final state.

4.1.9 V-CRC-VUL-009: Multiple Collusion Opportunities

Severity	Low	**Commit**	354bb0b
Type	Collusion	**Status**	Acknowledged
File(s)	`BaseReactor.sol`		
Location(s)	N/A		
Confirmed Fix At	N/A		

A reactor defines several entities who interact during the course of a cross-chain order, including the oracle, the filler and the swapper. There are several opportunities for these entities to collude which can be difficult for users to detect and can have a range of negative impacts.

Impact Below are some impacts of the collusion between the above entities:

1. Swapper + Oracle Collusion: A user can keep inputs and outputs (possibly minus some fee) if an order is filled as the oracle will never mark it as proven.
2. Filler + Oracle Collusion: A filler can keep inputs and outputs (possibly minus some fee) as an order will be fraudulently marked as proven.
3. Filler + Swapper Collusion: Can make a filler appear honest without requiring orders to be filled.
4. Filler + Swapper + Oracle Collusion: Can make an oracle and filler appear honest without requiring orders to be filled.

Recommendation Consider documenting signs of collusion between various entities to help users identify instances where it occurs. Also consider monitoring and publishing instances of collusion so that colluding oracles in particular can be blacklisted.

Developer Response The developers have acknowledged this issue.

4.1.10 V-CRC-VUL-010: Bitcoin Token with Unknown AddressType can be Verified

Severity	Low	**Commit**	354bb0b
Type	Data Validation	**Status**	Fixed
File(s)	`BitcoinOracle.sol`		
Location(s)	_bitcoinScript		
Confirmed Fix At	`2af9cd8`		

The Cross-cats Bitcoin Oracle makes use of the Bitcoin prism project to validate that order outputs are filled. As part of this process, it extracts some bitcoin "token" identifier about the expected behavior of the transfer. This information includes the `AddressType`, which is an identifier used to construct the script that is expected to fill an order. As shown below, however, it is possible that the token's `AddressType` is invalid as valid AddressTypes correspond to the integer values 1-5. While the remainder of the integer values do not correspond to a script type, it should be noted that they can still be proven. In this case, the returned bitcoin script is simply empty, allowing anyone to prove that they have filled the order with a transaction output that has no script (creating a UTXO that can be spent by anyone).

```
function _bitcoinScript(bytes32 token, bytes32 scriptHash) internal pure returns (
    bytes memory script) {
    // Check for the Bitcoin signifier:
    if (bytes30(token) != BITCOIN_AS_TOKEN) revert BadTokenFormat();

    // Load address version.
    AddressType bitcoinAddressType = AddressType(uint8(uint256(token)));

    return BtcScript.getBitcoinScript(bitcoinAddressType, scriptHash);
}
```

Snippet 4.8: Definition of `_bitcoinScript`

Impact This behavior is likely non-obvious and would likely not benefit either a swapper or filler as the swapper would not receive funds and the filler would have to put their funds at risk at least for a short time (but likely would be able to claim the unspent UTXO first).

Recommendation We would recommend implementing one of the following:

1. Consider implementing a whitelist for bitcoin tokens so that users cannot swap invalid tokens.
2. If a whitelist is unacceptable, consider always verifying invalid bitcoin tokens (i.e. tokens with an empty script) and placing sufficient warning of this behavior for users submitting swaps to reduce the likelihood of mistakes.

Developer Response The developers have implemented this recommendation.

4.1.11 V-CRC-VUL-011: Order Dispute can be Frontrun

Severity	Warning	Commit	354bb0b
Type	Frontrunning	Status	Acknowledged
File(s)	`BaseReactor.sol`		
Location(s)	dispute()		
Confirmed Fix At	N/A		

If an order is initiated by a filler but the order outputs are never paid out to the recipient for the outputs, a challenger can call the `dispute()` function after the fill deadline has passed to prevent the filler from receiving the order inputs. The challenger provides collateral that will be lost if the outputs are proven to have been delivered, but will receive a portion of the filler's collateral if the order can't be proven to be filled.

Since the challenger will receive funds if the challenge is successful, users are incentivized to challenge fraudulent transactions. Normally a challenger will have to verify that the order hasn't been filled to avoid losing their collateral. However, users can attempt to frontrun calls to `dispute()` rather than perform the verification themselves.

```
1    function dispute(
2        OrderKey calldata orderKey
3    ) external {
4        bytes32 orderKeyHash = _orderKeyHash(orderKey);
5        OrderContext storage orderContext = _orders[orderKeyHash];
6
7        // Check if order is claimed and hasn't been challenged:
8        if (orderContext.status != OrderStatus.Claimed) revert WrongOrderStatus(
    orderContext.status);
9        // If 'orderContext.status != OrderStatus.Claimed' then there are 2 cases:
10       // 1. The order hasn't been claimed.
11       // 2. We are past the claim state (disputed, proven, etc).
12       // As a result, checking if the order has been claimed is enough.
13
14       // Check if challenge deadline hasn't been passed.
15       if (uint256(orderKey.reactorContext.challengeDeadline) < block.timestamp)
    revert ChallengeDeadlinePassed();
16
17       // Later logic relies on  orderContext.challenger != address(0) if
    orderContext.status = OrderStatus.Challenged.
18       // As a result, it is important that this is the only place where these
    values are set so we can easily audit if
19       // the above assertion is true.
20       orderContext.challenger = msg.sender;
21       orderContext.status = OrderStatus.Challenged;
22
23       // Collect bond collateral.
24       // CollateralToken has already been entered so no need to check if it is a
    valid token.
25       SafeTransferLib.safeTransferFrom(
26           orderKey.collateral.collateralToken,
27           msg.sender,
28           address(this),
```



```
            orderKey.collateral.challengerCollateralAmount
        );

        emit OrderChallenged(orderKeyHash, msg.sender);
    }
```

Snippet 4.9: Definition of `dispute()`

Impact Normally frontrunning a call to `dispute()` without verifying whether the outputs have been delivered is risky because the original challenger may have made a mistake. However, if an address is known for correctly challenging fraudulent transactions, people may be able to confidently attempt to frontrun calls to `dispute()` from that address. In such a situation, the challenger that did the work of checking if the outputs have been delivered may not receive any reward for the challenge.

Additionally, if one of the inputs or the collateral token has a callback, a malicious user can frontrun the dispute to hold the funds hostage and try to extort the original user for more funds in exchange for their inputs.

Recommendation Include a warning about the potential for frontrunning calls to `dispute()` in the documentation, and recommend that users who intend to make many challenges should use different addresses to avoid being targeted by frontrunners.

Developer Response The developers have acknowledged this issue.

4.1.12 V-CRC-VUL-012: Read-Only Reentrancy in proveOrderFulfilment

Severity	Warning	**Commit**	354bb0b
Type	Reentrancy	**Status**	Won't Fix
File(s)	`BaseReactor.sol`		
Location(s)	proveOrderFulfilment		
Confirmed Fix At	N/A		

In the `proveOrderFulfilment` function, developer comments indicate that a call to the `localOracle` is untrusted and should be protected against traditional and read-only reentrancies. During our review, the Veridise security team noted that the code did protect against reentrancy attacks but does not properly protect against read-only reentrancies. This is because the order has been eagerly marked as proven *before* the protocol knows it is proven. Calls to `getOrderContext` will therefore incorrectly return that the order is proven.

```
function proveOrderFulfilment(OrderKey calldata orderKey, bytes calldata
    executionData) external {
    ...
    // Only allow processing if order status is either claimed or challenged.
    if (status != OrderStatus.Claimed && status != OrderStatus.Challenged) {
        revert WrongOrderStatus(orderContext.status);
    }

    // Immediately set order status to proven. This causes the previous line to fail.
    // This acts as a LOCAL reentry check.
    orderContext.status = OrderStatus.Proven;

    // The following call is a external call to an untrusted contract. As a result,
    // it is important that we protect this contract against reentry calls, even if
    read-only.
    if (!IOracle(orderKey.localOracle).isProven(orderKey.outputs, orderKey.
    reactorContext.fillDeadline)) {
        revert CannotProveOrder();
    }

    ...
}
```

Snippet 4.10: Snippet from `proveOrderFulfilment()`

Impact If a local oracle relied on a reactor to help it determine whether a group of outputs were proven, it could receive incorrect information.

Recommendation Consider introducing an intermediate state while an order's proof is being obtained.

Developer Response The developers have indicated that they do not plan to fix this issue.

4.1.13 V-CRC-VUL-013: Filler can Lose Funds when Purchasing Order

Severity	Warning	**Commit**	354bb0b
Type	Logic Error	**Status**	Fixed
File(s)	`BaseReactor.sol`		
Location(s)	purchaseOrder		
Confirmed Fix At	N/A		

A filler can specify a period of time during which an order may be purchased from them. Should someone purchase an order, they pay the collateral amount *and* the discounted order inputs to the old filler as shown below. Notably, however, the discounted price does not include the governance fee which is not "paid" until the order is filled. Therefore, when an order is purchased, the purchaser will earn the order inputs minus the discounted purchase price minus the governance fee.

```
1  function purchaseOrder(OrderKey calldata orderKey, bytes calldata fillerData, uint256
       minDiscount) external {
2      ...
3
4      // Collateral is paid for in full.
5      address collateralToken = orderKey.collateral.collateralToken;
6      uint256 collateralAmount = orderKey.collateral.fillerCollateralAmount;
7      // No need to check if collateral is valid, since it has already entered the
       contract.
8      SafeTransferLib.safeTransferFrom(collateralToken, msg.sender, oldFillerAddress,
       collateralAmount);
9
10     // Transfer the ERC20 tokens. This requires explicit approval for this contract
       for each token.
11     // This is not done through permit.
12     // This function assumes the collection is from msg.sender, as a result we don't
       need to specify that.
13     _collectTokensFromMsgSender(orderKey.inputs, oldFillerAddress,
       oldOrderPurchaseDiscount);
14
15     ...
16 }
```

Snippet 4.11: Snippet from `purchaseOrder()`

Impact In cases where the discount is less than the governance fee, it will be impossible for the purchaser to make a profit which is likely to be non-obvious to purchasers. Fillers are also incentivized to seek out such arrangements as they will receive more money in this case.

Recommendation Consider requiring that the order discount be greater than the governance fee when an order is available for purchase.

Developer Response The developers have documented this risk for order purchasers.

4.1.14 V-CRC-VUL-014: Parsed BitcoinAddress may be Invalid

Severity	Warning	**Commit**	354bb0b
Type	Data Validation	**Status**	Fixed
File(s)	`BtcScript.sol`		
Location(s)	getBitcoinAddress		
Confirmed Fix At	`2af9cd8`		

The `BtcScript` library provides utilities to interact with Bitcoin scripts. One such utility is the ability to extract a `BitcoinAddress` from the script which corresponds to an identifier of the UTXO recipient. To do so, it first attempts to identify the script being used and once the script is known will extract script-specific information about the recipient. There are a few corner cases however where the information returned is inaccurate:

1. The following function determines if the script corresponds to P2PKH or P2SH by checking the first byte and length of the script but this does not guarantee that the script is correct. Rather the `decode*` functions will check the individual opcodes to ensure they are consistent with the script type. If they are not, 0 will be returned as the implementation hash to indicate that decoding failed and yet the `AddressType` field of the `BitcoinAddress` will correspond to the expected script rather than unknown.
2. The `decodeWitnessProgram` function is used to extract information about scripts that use witnesses, including the version and length of the witness. While the P2WPKH and P2WSH address types correctly checks that both the version and length of the witness match the expected value, the P2TR address type does not. This could allow a script to be marked as P2TR even if it has the wrong witness length.

```
1  function getBitcoinAddress(bytes calldata script) internal pure returns(
       BitcoinAddress memory btcAddress) {
2      // Check if P2PKH
3      bytes1 firstByte = script[0];
4      if (firstByte == OP_DUB) {
5          if (script.length == P2PKH_SCRIPT_LENGTH) {
6              btcAddress.addressType = AddressType.P2PKH;
7              btcAddress.implementationHash = decodeP2PKH(script);
8              return btcAddress;
9          }
10     } else if (firstByte == OP_HASH160) {
11         if (script.length == P2SH_SCRIPT_LENGTH) {
12             btcAddress.addressType = AddressType.P2SH;
13             btcAddress.implementationHash = decodeP2SH(script);
14             return btcAddress;
15         }
16     } else {
17         // This is likely a segwit transaction. Try decoding the witness program
18         (int8 version, uint8 witnessLength, bytes32 witPro) = decodeWitnessProgram(
       script);
19         if (version != -1) {
20             if (version == 0) {
21                 if (witnessLength == 20) {
22                     btcAddress.addressType = AddressType.P2WPKH;
23                 } else if (witnessLength == 32) {
```



```
                    btcAddress.addressType = AddressType.P2WSH;
                }
            } else if (version == 1) {
                btcAddress.addressType = AddressType.P2TR;
            }
            btcAddress.implementationHash = witPro;
            return btcAddress;
        }
    }
}
```

Snippet 4.12: Definition of `getBitcoinAddress`

Impact The above corner cases could provide incorrect information about a script.

Recommendation We would recommend that the developers ensure the behavior is correct with respect to these corner cases

Developer Response The developers have implemented the above recommendation.

4.1.15 V-CRC-VUL-015: One Chain ID may map to Multiple Escrows

Severity	Warning	**Commit**	354bb0b
Type	Logic Error	**Status**	Fixed
File(s)	GeneralisedIncentivesOracle		
Location(s)	setRemoteImplementation		
Confirmed Fix At	`ca6bef7`		

The `setRemoteImplementation` function can be used by contract owners to whitelist an `IncentivizedMessageEscrow` on another chain that may communicate with this contract. The `IncentivizedMessageEscrow` contract, however, has the invariant that only a single remote implementation may be associated with a chain. This function can break that invariant, however, if multiple "chain identifiers" map to a single chainId.

```
function setRemoteImplementation(
    bytes32 chainIdentifier,
    uint32 blockChainIdOfChainIdentifier,
    bytes calldata implementation
) external onlyOwner {
    //  escrow.setRemoteImplementation does not allow calling multiple times.
    escrow.setRemoteImplementation(chainIdentifier, implementation);

    _chainIdentifierToBlockChainId[chainIdentifier] = blockChainIdOfChainIdentifier;
    emit MapMessagingProtocolIdentifierToChainId(chainIdentifier,
    blockChainIdOfChainIdentifier);
}
```

Snippet 4.13: Definition of `setRemoteImplementation`

Impact An invariant of the `IncentivizedMessageEscrow` could be broken which could be an indicator of malicious activity.

Recommendation Consider maintaining a reverse mapping from the `blockChainIdOfChainIdentifier` to the `chainIdentifier` and enforce that only a single `chainIdentifier` can be associated with a chainId.

Developer Response The developers have applied the above recommendation.

4.1.16 V-CRC-VUL-016: Function Missing Override

Severity	Info	**Commit**	354bb0b
Type	Maintainability	**Status**	Fixed
File(s)	`ExclusiveOrder.sol`		
Location(s)	validate()		
Confirmed Fix At	`e41e4ae`		

The `ExclusiveOrder` contract implements the `IPreValidation` interface, but doesn't mark the validate function with `override` .

```
function validate(bytes32 key, address initiator) external view returns (bool)
```

Snippet 4.14: Signature of `ExclusiveOrder.validate()`

```
/**
 * @notice Validation logic that adds selective order validation logic.
 */
interface IPreValidation {
    /**
     * @notice Validate if the order shall progress.
     * @param validationKey Can contain various encoded logic for validation.
     * @param initiator Caller of initiate on the reactor. May or may not be the
    destination address (filler).
     */
    function validate(bytes32 validationKey, address initiator) external view returns
     (bool);
}
```

Snippet 4.15: Definition of `IPreValidation`

Impact The missing `override` could cause developer confusion in the future.

Recommendation Mark this function as `override`

Developer Response The developers have implemented the above recommendation.

4.1.17 V-CRC-VUL-017: Protocol Can Introduce Reentrancies into Other Protocols

Severity	Info	**Commit**	354bb0b
Type	Reentrancy	**Status**	Fixed
File(s)	`BaseReactor.sol`		
Location(s)	proveOrderFulfilment, optimisticPayout, purchaseOrder		
Confirmed Fix At	N/A		

It is very common for DApps to build logic that interacts with other blockchain applications. Doing so comes with risks as the developer must ensure they understand the trust assumptions of the protocol that they're building on. We therefore often recommend that protocols document certain design decisions to ensure others are aware of potentially unsafe interactions. In the case of cross-cats, there are unsafe calls exist in `proveOrderFulfilment`, `optimisticPayout`, `purchaseOrder` that could introduce a reentrancy vulnerability into another project as shown below. It should be noted that cross-cats itself has properly protected itself against these unsafe calls but others could be affected.

```
1  function purchaseOrder(OrderKey calldata orderKey, bytes calldata fillerData, uint256
       minDiscount) external {
2      ...
3
4      // Check if there is an identifier, if there is execute data.
5      if (oldIdentifier != bytes32(0)) {
6          FillerDataLib.execute(identifier, orderKeyHash, fillerData[fillerDataPointer
       :]);
7      }
8
9      emit OrderPurchased(orderKeyHash, msg.sender);
10 }
```

Snippet 4.16: Location of an unsafe call in `purchaseOrder`

Impact Locations where unsafe calls are made could be used to perform a reentrancy attack against another protocol that builds on cross-cats.

Recommendation We would highly recommend documenting this so that other developers are aware of the risks.

Developer Response The developers have implemented the above recommendation.

