PUBLIC

Code Assessment

of the MangroveOrder Smart Contracts

December 13, 2022

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Findings 10
6 Resolved Findings 11
7 Notes 17

1 Executive Summary

Dear Jean,

Thank you for trusting us to help Giry SAS with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of MangroveOrder according to Scope to support you in forming an opinion on their security risks.

Giry SAS implements a peripheral contract for the Mangrove core system which allows users to submit Good-till-cancelled orders and Fill-or-kill orders.

The most critical subjects covered in our audit are functional correctness, absence of reentrancy possibilities, access control, handling of funds, and accounting. We have uncovered some important bugs. Regarding functional correctness, we uncovered a bug where the gas price for an updated order is calculated and submitted incorrectly. Regarding accounting, we have uncovered a vulnerability affecting the order updates which can allow an attacker to steal funds from Mangrove core system. However, the impact of the vulnerability is not big since it is not expected that an attacker can steal a significant amount. Moreover, as far as internal accounting is concerned, if an updated order requires less provision than before, the provision is not refunded to the end users. All the aforementioned issues were addressed in the second iteration.

The general subjects covered are code complexity, use of uncommon language features, unit testing, documentation, specification, gas efficiency, trustworthiness and error handling. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	3
• Code Corrected	2
• Specification Changed	1
Low-Severity Findings	5
• Code Corrected	5

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the MangroveOrder repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	18 October 2022	16f97b3dbd6a8c86ab67bc8be9008c1877c7fd58	Initial Version
2	5 December 2022	ab596cd1afbdd828eb4d02bc680b6825ca68172e	Fixes
3	12 December 2022	c55978eecd5c74be8567b6759945727d66b6cd05	Updated Fixes

For the solidity smart contracts, the compiler version `0.8.13` was chosen.

In scope is the `MangroveOrder` contract and all the contracts and library this contract uses. More specifically:

- `periphery/MangroveOrder.sol`
- `MgvLib.sol`
- `strategies/utils/AccessControlled.sol`
- `strategies/routers/AbstractRouterStorage.sol`
- `strategies/routers/AbstractRouter.sol`
- `strategies/routers/SimpleRouter.sol`
- `strategies/MangroveOfferStorage.sol`
- `strategies/MangroveOffer.sol`
- `strategies/offer_forwarder/abstract/Forwarder.sol`

2.1.1 Excluded from scope

Everything not included in scope. Part of the code in scope was automatically generated. The correctness of the generation process as well as the end result of that process is considered out-of-scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Giry SAS offers a periphery contract (`MangroveOrder`) for the Mangrove-core system. The contract stands as an intermediate component between the end-users and the core system meaning that users

should only interact with the `MangroveOrder`, while the contract itself is seen as a maker for the core system. The contract can act both as a taker and as a maker. As a taker, it creates an order that will try to match with some of the orders in the orderbook. As a maker, it posts an order to the orderbook and implements the appropriate logic for the hooks made by the core system to provide the liquidity needed by the posted order.

The orders implemented by the contract are:

- **Good till canceled (GTC):** An order which is originally a market order i.e., a buy or a take order, which is then posted to the order book (if not completely filled). For a GTC order which wants an amount `a_goal` and price `p_goal`, if the order is partially filled for an amount `a_now` for which `a_now < a_goal` and price `p_now` for which `p_now < p`, then the order posted will be with price `p_later` so that `a_now*p_now + (a_goal - a_now)*p_later = p_goal*a_goal`. Intuitively, if a user was able to buy an amount `a_now` for a cheaper price than expected, they can afford to pay a greater price for the remaining amount. Note that we ignore the fees for simplicity.
- **Fill or kill (FOK):** An order that is either fully filled right away or is ignored in its entirety by the system.
- **Partially filled:** An order which can be partially filled but never posted.

The contract exposes the following functions to the users:

- **take:** It implements the market order part of an order for the GTC orders and the other two orders. The `MangroveOrder` contract withdraws the amount the taker wants to give and calls `Mangrove.marketOrder`. After the order has been matched with as many orders from the orderbook as possible, its completeness is checked and the leftover amount to be given is returned to the taker's reserve as well as the received amount. Then, for GTC orders, the remaining order is posted. For FOK orders, if the order hasn't been fully filled the transaction is reverted. Finally, for all types of orders that haven't been already reverted, any remaining native ETH is sent back to the caller.
- **updateOffer:** The owner of an order can update all its parameters.
- **retractOffer:** The owner of an order can retract their offer. The native ETH which is sent to `MangroveOrder` is then sent to the caller.
- **withdrawToken:** It transfers the specified token from the msg.sender's reserve to an arbitrary receiver.

Users can define a different address/contract in which they can store the funds to be used by the MangroveOrder. For the reserve management:

- **setRouter:** The maker sets an arbitrary address as its reserve.
- **approvePooledMaker:** The reserve approves the maker to use its address as a reserve.
- **revokePooledMaker:** The reserve revokes the approval from the maker to use its address as a reserve.

Since `MangroveOrder` can act as a maker it implements the `IMaker` interface required by Mangrove:

- **makerExecute:** The hook called by the Mangrove core system with which the maker provides liquidity promised by their order. It checks to make sure the order hasn't expired. It transfers the inbound tokens to the user's reserve and sends the required amount to the `MangroveOrder` contract to be withdrawn by Mangrove.
- **makerPosthook:** The hook executed after the order has been executed. It gives a chance for the order to update its state. If the order was executed successfully and it was not fully filled, the order is reposted with updated amounts. If the order reverted at any point, then it tries to estimate how much penalty was deducted by the provision and credits the rest to the user. It is

important to note that in case the gas provided is not sufficient, the remainder of the provisioned amount will not be credited.

Finally, `MangroveOrder` exposes some admin functions:

- **withdrawFromMangrove:** It withdraws all the available ETH for the `MangroveOrder` contract stored in Mangrove core.
- **activate:** It approves Mangrove-core and the router to transfer money from `MangroveOrder`.
- **setAdditionalGasreq:** Sets the minimum additional gas requirement for the orders created by `MangroveOrder`.

An important component of `MangroveOrder` is the router (`SimpleRouter`). The router is responsible for transferring funds from the end-user's reserves to the `MangroveOrder` and vice versa.

2.2.1 Trust Model and Roles

The system defines the following roles:

- The admin: they have a privileged role in the system and they are fully trusted to not take actions that can harm the system or the users.
- Mangrove core: The Mangrove core contract can make specific calls (see `IMaker`) to the contract. It is assumed that Mangrove is safe and will not pass malicious data to the hooks.
- Normal users: They create orders and update or remove their orders.

2.3 Version 2

In the second iteration of the codebase the specification of `MangroveOrder` has been updated. In particular:

1. The orders submitted through `MangroveOrder` are now added to the order book using the price the user specified as a taker.
2. Users can no longer arbitrarily set the gas requirements for the orders they update.
3. Orders can specify an absolute expiration date and not a time-to-live for the order.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	2

- Locked Refunded Provision Code Corrected
- Wrong Calculation of Locked Provision Code Corrected

Medium-Severity Findings	3

- Expiration Date Cannot Be Updated Code Corrected
- Underflow in postRestingOrder Specification Changed
- Users Can Steal Funds From MangroveOrder Code Corrected

Low-Severity Findings	5

- Inaccurate Comment Code Corrected
- Missing Natspec Code Corrected
- Redundant pragma abicoder v2 Code Corrected
- Setting Expiration Date Code Corrected
- Forwarder.provisionOf Calculation Is Wrong Code Corrected

6.1 Locked Refunded Provision

Correctness High Version 1 Code Corrected

When a maker submits an order to the Mangrove orderbook, they need to provide some ETH, also known as the provision, to compensate the takers in case the `makerExecute` hook reverts. A maker can update their offer by calling `Forwarder.updateOffer`. Note that at this point a maker can update most of the parameters of the order including `gasreq`, i.e. the gas required for the `makerExecute` hook to execute. A maker could reduce the gas requirements meaning that some provision will be refunded to them. `Forwarder.updateOffer` does not handle this refunding (the `ownerData.weiBalance` is not updated) and Mangrove system only sees `MangroveOrder` as a maker. This means that the refunded amount is essentially lost for the end-user of the `MangroveOrder`. Note that if the provision needs to be increased again, the end-user must provide extra ETH.

Code Corrected:

In the current implementation, the provision can only be increased therefore no funds are locked.

6.2 Wrong Calculation of Locked Provision

Correctness High Version 1 Code Corrected

When a user updates their offer through `Forwarder.updateOffer`, `MangroveOrder` tries to calculate the new gas price by calling `deriveGasprice`. The gas price depends on the total provision available for this order. That is the sum of the extra provision attached which is stored in `args.fund` and the already locked provision. Currently, the locked amount is calculated with the following snippet:

```
vars.offerDetail.gasprice() * 10 ** 9 * args.gasreq + vars.local.offer_gasbase()
```

This formula is wrong for two reasons:

1. It depends on `args.gasreq` which is the updated gas requirement of the order as passed by the user.
2. There are parentheses missing around `args.gasreq + vars.local.offer_gasbase()`, as this entire term should be multiplied by the gas price.

This miscalculation can have multiple consequences:

1. Can allow users to steal funds (see relevant issue).
2. An order can be submitted with smaller gasprice since the calculated total provision is too small.

Code Corrected:

`Forwarder.updateOffer` has been updated. Currently, users can only increase the provision for an order. Users cannot determine `args.gasreq` as it is set to be equal to the `offerGasreq()`. It is important to notice that `offerGasreq()` is not constant but depends on the configuration of the `MangroveOrder` and in particular the gas requirements of the router.

6.3 Expiration Date Cannot Be Updated

Design Medium Version 1 Code Corrected

A user can update most of the offer details by calling `Forwarder.updateOffer`. However, the expiration date cannot be changed. In order to change the expiration date of an order, one must retract it and submit a new one.

Code Corrected:

`MangroveOrder.setExpiry` has been added to allow users to update the expiration date of the order.

6.4 Underflow in postRestingOrder

Correctness Medium Version 1 Specification Changed

Once the market order part of GTC order has been filled as much as possible, the remaining amount the user wants to trade is put into a resting order. Note that if `fillWants == true`, then the Mangrove engine will have stopped matching the order either when it is fully filled, there are no more orders on the books, or when the total average price of the order would fall below the threshold of the ratio between the order's initial wants and gives. Hence, if the matching stops before the order's wants are fully filled, we are guaranteed not to have given away more than the order initially had (else the total average price would be below what we initially wanted).

However, if `fillWants == false`, this condition no longer holds. The order can receive arbitrarily many tokens before giving away all the tokens it has to give away. As the price of a trade is defined by the maker, there could be orders on the books which give away arbitrarily many tokens for a very low price. Hence, the user can receive more tokens in the market order part of the trade than they were expecting to. As such, `res.takerGot + res.fee` can exceed `tko.takerWants` despite only having partially filled the order.

When we go to post a resting order, the following code is executed:

```
res.offerId = _newOffer(
  OfferArgs({
    outbound_tkn: outbound_tkn,
    inbound_tkn: inbound_tkn,
    wants: tko.makerWants - (res.takerGot + res.fee), // tko.makerWants is before slippage
    gives: tko.makerGives - res.takerGave,
    gasreq: offerGasreq() + additionalGasreq, // using default gasreq of the strat + potential admin defined increase
    gasprice: 0, // ignored
    pivotId: tko.pivotId,
    fund: fund,
    noRevert: true, // returns 0 when MGV reverts
    owner: msg.sender
  })
);
```

When the `wants` for the resting order are calculated, an underflow can occur in the case described above, as the market order part of the GTC order could have received arbitrarily many tokens. As Solidity 0.8.10 is used, this will simply revert the transaction, but will unnecessarily prevent the user from completing their trade.

Specification Changed:

Currently, the order is posted with the same price as the taker originally wanted. Thus, the issue has been mitigated.

Giry SAS replied:

> this problem made use reevaluate our specification: requiring the (instant) market order and the (asynchronous) maker order to respect a limit **average** price is not well defined. In some cases this would lead the maker order to be posted for a 0 price. We decided to change the specification and post the maker order at the price initially set by the taker for the market order (irrespectively of the obtained price).

6.5 Users Can Steal Funds From `MangroveOrder`

Security **Medium** **Version 1** **Code Corrected**

The core Mangrove system maintains the `balanceOf` mapping which stores how much ETH is available for each maker to be used as a provision for their orders. Importantly, the `MangroveOrder` contract is seen as **one single** maker by the system, even though there might be **many** end users creating their orders through it. Let us assume that at some point the balance of `MangroveOrder` is positive and an attacker has already submitted an order. It is possible as we show in another issue that there might be some non-claimable balance since `updateOrder` does not handle refunds. An attacker can steal money from mangrove by employing any of the following two vectors:

1. **Updating an order without sending funds:**

   - The attacker calls `Forwarder.updateOrder` for their order with `msg.value == 0` and they increase the gas requirement of their order.
   - This means that `args.fund == 0` so gas price will remain the same, however, the total provision needed has been increased as the gas requirements have been increased!
   - At this point `MGV.updateOffer` is called with `msg.value == 0`.

- Mangrove core does not perform any check if there are enough funds attached to the call since it relies on the `balanceOf` mapping by calling `debitWei`.
- Mangrove core uses the amount stored in `balanceOf` for the extra provision.
- The attacker now retracts the order and withdraws the provision of the order which includes the stolen amount.

2. **Updating an order by attaching funds:**

   - The attacker calls `Forwarder.updateOrder` for their order with `msg.value != 0` and they increase the gas requirement of their order.
   - Since funds have been attached to the transaction, the gas price will be recalculated.
   - The new provision at this point is calculated wrongly since the `provision` parameter passed to `derivePrice` depends on `args.gasreq` which represents the updated gas requirements of the offer and not `vars.offerDetail.gasreq()`. Note that `args.gasreq` can be freely set by the users so arbitrarily large value could be passed. As a result, the new gas price is greater than it should be but the extra funds passed are not enough to cover for the extra provision needed by the offer.
   - Mangrove core uses the amount stored in `balanceOf` for the extra provision.
   - The attacker now retracts the order and withdraws the provision of the order which includes the stolen amount.

A similar attack can be performed when some of the global parameters change, which could result in inaccurate accounting of provisions. If the `gasbase` of the token pair related to an order changes in the core mangrove system, calling `updateOffer` can result in an increased (or decreased) provision without providing any additional funds. This will credit (or debit) funds to the *MangroveOrder* contract which aren't attributed to any user. In particular, if the global gas price is increased, calling `updateOffer` of Mangrove core with an unchanged gasprice which is lower than the new global gas price, the mangrove core system will set the gas price higher without receiving any funds. This again changes the balance of the *MangroveOrder* contract, without attributing it to any individual user. While `_newOffer` and `_updateOffer` in *Forwarder* have checks to make sure the offer's gas price is higher than the global gas price, `__posthookSuccess__` in *MangroveOffer* does not. Hence, if the global gas price changes, then an order is partially filled and attempts to repost, its provision will be increased with no additional submitted funds. While the amounts of funds are small, it is conceivable that a malicious user could be able to exploit a change in the global gas price or the `gasbase` in order to steal funds.

It is important to note that this issue cannot result in users losing funds since the excessive provision which can be stolen cannot be claimed by any specific user. In the normal case, no excessive provision should be available. Therefore, it is expected the amount that can be stolen to be low. Hence, we consider the issue as medium severity.

Code partially corrected:

The issue has been addressed in multiple different ways:

1. In the current implementation there shouldn't be unallocated users' funds in Mangrove core.
2. Users can only increase the provision of an order using `MangroveOrder.updateOrder`, not decrease it. Hence, they must provide additional provision and can not submit orders which could make use of funds that are already stored in the Mangrove core.
3. The `__posthookSuccess__` uses `Forwarder._updateOffer`.

6.6 Inaccurate Comment

Design Low Version 1 Code Corrected

In `MangroveOrder.checkCompleteness`, the following is mentioned:

> // when fillWants is true, the market order stops when *takerWants* units of *outbound_tkn* have been obtained;

However, this comment is inaccurate since part of the `takerWants` goes to cover the fees, so not the full `takerWants` amount can be obtained.

In `AbstractRouter.push`, the return value is described as follows:

> ///@return pushed fraction of *amount* that was successfully pushed to reserve.

However, for tokens with fees, provided the `TransferLib` is used, the whole amount will always be reported.

Code Corrected:

The comments have been updated.

6.7 Missing Natspec

Design Low Version 1 Code Corrected

The Natspec is missing in the following cases:

- For `AbstractRouter.bind`, the `maker` parameter.
- For `AbstractRouter.unbind`, the `maker` parameter.
- For `SimpleRouter.__pull__`, the `strict` parameter.
- For `IOfferLogic.OfferArgs`, the `gasprice` field.

Code Corrected:

The Natspec has been added to the respective functions.

6.8 Redundant `pragma abicoder v2`

Design Low Version 1 Code Corrected

Many contracts include the `pragma abicoder v2` directive. However, for solidity 0.8 the abicode v2 is the default one, so the pragma is redundant.

Code Corrected:

The `pragma` has been removed from most of the contracts.

6.9 Setting Expiration Date

Design | Low | Version 1 | Code Corrected

A user can define the time-to-live of a resting order submitted through `MangroveOrder` by specifying the `TakeOrder.timeToLiveForRestingOrder`. It is important to note that an order can remain in the mempool for a long time before it's executed. Specifying an explicit expiration date instead of the time-to-live might be more convenient for users since it's independent of the time it takes for a transaction to be included in a block.

Code Corrected:

The expiration date is now absolute and no longer relative to the time the transaction is added to the blockchain.

6.10 `Forwarder.provisionOf` Calculation Is Wrong

Design | Low | Version 1 | Code Corrected

As its natspec suggests `Forwarder.provisionOf` computes the amount of native tokens that can be redeemed when deprovisioning a given offer. However, this is not true. In `MgvOfferMaking.retractOffer`, the provision is calculated as follows:

```
provision = 10 ** 9 * offerDetail.gasprice() //gasprice is 0 if offer was deprovisioned
  * (offerDetail.gasreq() + offerDetail.offer_gasbase());
```

The important part to notice is that provision depends on `offerDetail.offer_gasbase()`.

This is not the same for `Forwarder.provisionOf` where the provision is calculated as follows:

```
provision = offerDetail.gasprice() * 10 ** 9 * (local.offer_gasbase() + offerDetail.gasreq());
```

Here, the provision depends on `local.offer_gasbase()` instead of `offerDetail.offer_gasbase()`.

Code Corrected:

The provision is now calculated using the `offerDetail.offer_gasbase()`.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Updating Approvals on Order Update

Note Version 1

A user can update their orders by using `Forwarder.updateOffer`. It is important for users to remember that, in case the `makerExecute` hook to their order fails, they will have to reimburse the taker. A reason for an order to fail is that there is not enough allowance given to the router to transfer funds from the maker's reserve to `MangroveOrder` contract. This is highly likely to happen after a user updates their offer by having it give more funds to the taker.