PUBLIC

Code Assessment

of the Mangrove Smart Contract

March 02, 2022

Produced for

by

Contents

1 Executive Summary

Dear Adrien,

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

Giry implements an offer book exchange supporting markets between two assets. Makers create and takers consume offers which are promises of makers to provide the offered token at a certain price. To ensure the executability of offers, makers must deposit ether for gas reimbursements on failure.

The most critical subjects covered in our audit are functional correctness, access control, precision of arithmetic operations, front-running and signature handling. Security regarding most of the aforementioned subjects is high. Security of signature handling is basic due to possible ECDSA malleability, see ECDSA Signature Malleability. Security of front-running is good but keepers could lose funds to rogue makers unexpectedly due to unawareness of the exact functionality of sniping, see No Protection for Keepers.

The general subjects covered are unit testing, documentation, specification, gas efficiency 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	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	3
• Code Corrected	2
• Risk Accepted	1
Low-Severity Findings	11
• Code Corrected	7
• Risk Accepted	2
• Acknowledged	2

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 for the `Mangrove` contract inside the private Mangrove-archive repository based on the documentation files. Chainsecurity confirms that code of the main Mangrove contract in the public repository at the listed commit is equal to the final code of the Mangrove contract as Version 3 in the private repo.

The table below indicates the code versions relevant to this report and when they were received.

Mangrove-archive

V	Date	Commit Hash	Note
1	28 June 2021	43e003314740adddf584f729856139f84feff813	Initial Version
2	14 July 2021	f122b2017da774638b49ae4272bdf1e2ca5866a2	After Intermediate Report
3	21 February 2022	b41725a596c760c52d66c18a8c6ae5904d51d74d	Updated code
4	02 March 2022	9bee9671a146483335c58bfff2f501d398599507	Final commit

Mangrove Public Repository

V	Date	Commit Hash	Note
1	21 February 2022	e15455d4937535e65209035ba05c2bc9579f6129	Corresponding version 3
2	02 March 2022	a0fff969396a3b1f7edac1a0f6ffff3af30b0ac7	Corresponding version 4

For the solidity smart contracts, the compiler version `0.7.6` was chosen. In the updated code the compiler version was updated to `0.8.10`.

2.1.1 Excluded from scope

The contracts in subfolder `LPcontracts` (renamed to `Strategies` in Version 3) are out of scope. The project makes use of the solpp preprocessor and the smart contracts contain solpp instructions. The review has been done on the code containing solpp instructions, the correct working of the solpp preprocessor to generate the correct solidity code is out of scope.

In version 3, preprocessing code has been removed from the core contracts which was added to the newly introduced MgvPack.sol which is out of scope and assumed to be correct.

3 System Overview

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

At the end of this report section we have added subsections for each of the changes accordingly to the versions. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Mangrove implements an offer book based exchange. Individual offer books exist for each market consisting of a `base` and a `quote` asset. Technically an offer book is a sorted doubly linked list of offers. Each offer promises an amount of the so-called `base` asset and requests a certain amount of the `quote` asset. `Makers` create these offers. `Takers` take these offers by executing a so-called order. During the execution of an order, the amount of the base quote is transferred to the maker first before the maker address is called to execute arbitrary code. During this call, the maker must do all actions necessary and make the amount of the base asset available for the exchange to collect.

Offers are just promises and the execution of an order may fail. When an offer fails e.g., because it failed to make available the amount of tokens to the exchange, the execution of the order is stopped. A penalty mechanism exists to incentivize makers to have working offers. Upon offer creation, the maker has to provide a so-called provision in Ether to cover for the gas costs should the transaction revert. If the offer succeeds, the provision is returned to the maker. When an offer fails, a part of the provision is given to the taker to compensate for his lost gas costs.

A callback to the maker at the end of an exchange allows the maker to update his offer.

The system is administrated by the governance which can add/remove or pause token pairs or change the parameters of the system.

The most important functions are:

Offer Creation:

`newOffer()`: This function allows anyone to create a new offer. Similarly `updateOffer` and `retractOffer` allow the maker to change and remove his existing offers respectively. The given amount must be above a certain threshold compared to its gas requirement multiplied by the market's `density` parameters.

General Market Order:

`marketOrder()`: A taker, who can be anyone, can call this function and specify the amount of the base and the quote asset they are willing to spend. The smart contract executes the order by iterating through the offer book until either the order is filled or the price of the remaining offers is too high or the end of the offer book has been reached. The order can be filled in two ways depending on the `fillWants` parameter. If this parameter is set to true, then, the order is filled if the total base asset (what the taker wants) has been obtained. Otherwise, the order is filled if the total quote asset (what the taker gives) has been sold.

Technically, the execution works as follows:

The taker initiates a so called order which consumes offers inside the offerbook. During execution, a `multiOrders` struct keeps track of the overall order state while a single order struct is used to keep track of data related to the execution of the current single offer being processed within the whole order execution. A reentrancy lock ensures that the market for this base/quote pair is locked during the actual execution. The order execution starts with the current best offer. The filling is executed recursively until either the order has been fulfilled (parameter dependent either the order fill or order gives limit has been reached), there is no offer left in the offer book or the price of the next offer will make the average price exceed what the taker is ready to pay across all offers that are being executed during the market order.

Wrapped in a fail-safe call (meaning any revert will be caught) the following steps are executed:

1. Transfer of the quote asset from the `Maker` to the `Mangrove` contract
2. Transfer of the quote asset from the `Mangrove` to the `Maker` address
3. Call of function `makerExecute()` passing the struct `singleOrder` as argument to the `Maker` address passing along `gasreq` amount of gas as defined by Offer. The gas amount used during this call including some overhead to handle the return data is recorded.
4. Check whether the call was successful
5. Transfer of the base asset from the `Maker` to the `Mangrove` contract

Any issue will lead to a revert of this inner call with an appropriate error message. This error message will be caught and handled appropriately by the execution flow.

Such failure reasons are:

Taker fault:

- `mgv/notEnoughGasForMakerTrade`: the taker provided insufficient gas
- `mgv/takerTransferFail`: `transferFrom()` of tokens from the taker to `Mangrove` failed

Maker fault:

- `mgv/makerRevert`: The execution of the call to `Maker.makerExecute()` failed
- `mgv/makerTransferFail`: `transferFrom()` of tokens from maker to Mangrove failed
- `mgv/makerReceiveFail`: `transfer()` to the maker from Mangrove failed

After the failsafe call completes, its return data is handled.

If the call was successful the amounts received/paid in the multi order struct are updated accordingly. If the call has been unsuccessful and the taker is at fault, the whole transaction is reverted.

If the order has been executed, either successfully or unsuccessfully due to a fault by the maker the offer is removed from the offer book. In case it was unsuccessful the order is deprovisioned. Note this step is skipped when the offer has not been executed due to the price not being acceptable. At the end, the status of the order is propagated to the caller of the function.

The next best offer is taken from the offer book and the struct single order is updated accordingly before `innerMarketOrder()` is called recursively.

In order to terminate the recursion, `internalMarketOrder()` first checks if one of the conditions to terminate has been reached. If so it updates the pointer to the current best order, disables the reentrancy lock and initiates the transfer of the tokens to the taker while applying the fee.

After the recursion has reached the bottom the post processing of the order is done while stepping out of the recursion. This means the offers are now handled in reverse order: While orders have been executed in order 1-2-3, the post processing happens in reverse order 3-2-1.

During the post execution inside the recursion the following is executed for each:

The `postExecute` callback on the maker's address is executed. This allows the maker to update their offers. Note that this is now possible as the lock on the market has been lifted (which would have prevented the update of offers). The code is indifferent of whether this call is successful or not. While the gas consumption of this post hook call to the maker contract is counted, it's only refunded to the taker when the maker offer failed. In case the offer was not executed successfully, the penalty is applied here. Applying the penalty means calculating the difference in gas used vs provision provided by the maker, the difference is refunded to the maker. The total penalty for the `taker` is accrued.

Outside of the recursion the accrued Penalty (if there is one) is sent to the `taker` and the call terminates.

Important to note, an offer that is used during the execution of an order is always consumed. Even when the order does not consume the whole amount available in the offer, the offer is taken out of the offer book. A callback at the end of the order execution allows the maker to reinstate the order.

Sniping:

`snipe()`: Instead of taking the currently best available offer for a token pair, using function `snipe` a taker can specify the exact `offerId` to execute.

This may be desired for multiple reasons, including:

- For Keepers: Executing offers that fail and collecting the gas provision is worthwhile and helps to keep the offer book clean.
- For Takers: Executing a specific offer may be worthwhile as the callback to the maker may require less gas compared to the currently available best offer.

Similarly `snipes()` allows to execute multiple offers at once. Note that in this case `takerWants` and `takerGives` are specified per offer and not over the whole order.

While sniping an offer may technically be possible with zero amounts for `taker.gives` / `taker.wants`, smart makers could easily make such orders succeed and keep the provision. Generally, to remove an offer one has to execute it either successfully or unsuccessfully. For this, the taker has to provide an amount of the token quoted upfront where the maker lets the transaction fail on his end.

Mechanism of the penalty system:

Whenever an offer fails to hold its promise, part of the provision put up by the maker is paid to the taker as penalty. This penalty should overcompensate the taker for his lost gas cost and hence incentives keepers to keep the offer book free of failing orders. The actual amount of gas the penalty accounts for depends on how many offers executed in this order have failed and is calculated as:

(gasreq + overhead_gasbase/n + offer_gasbase)

This aims to keep the actual penalty low while ensuring it overcompensates for the actual lost gas. E.g. if multiple order fail during a `marketOrder`, the base costs (think of e.g. the tx base fee and the overhead executing code of Mangrove) are split in their penalty. However for failing orders the maximum penalty is always at risk if keepers choose to snipe the order individually.

This gas amount to be compensated is then multiplied with the current global gasprice of Mangrove. As the provision (which was based on the gas price at the time the offer was created) may be insufficient to cover the penalty, the penalty is capped at the provision of the order.

Delegation of rights to take orders:

Takers may provide allowances on specific pairs, so approved addresses can take orders in their name. To do so, takers either give the approval directly on chain or craft a EIP712 signature which the approved address can use on chain to activate his allowance. The approved account can use this allowance on the specific pair to execute `marketOrderFor()` and `snipeFor/snipesFor()` in the name of the taker who gave the allowance.

When using the `permit()` functionality one has to respect the sequence of the signatures: Due to the ever incrementing nonce used the signatures generated by a particular taker have to be used on-chain in the right order according to their nonce.

3.1 Roles

Untrusted roles which can be executed by anyone:

Makers: Create offers into the offer book, hence provide the liquidity.

Takers: Take and execute offers of the offer book.

Approved addresses: Can take offers on behalf of takers that approved them. These addresses must be fully trusted by the respective taker.

Keepers: Keep the offer book clean by executing failing orders and collecting the provision.

Trusted roles:

Governance: Administrates the system. Can set the following parameters:

Market related:

- Adding a market: This constitutes adding a pair of ERC20 tokens as base and quote respectively. Note that only tokens with no special behavior are intended to be added. Moreover, adding a market includes setting the `fee`, `density`, `overhead_gasbase` and `offer_gasbase`.
- All the parameters for a market can be changed individually
- Deactivating a market
- Killing the entire system - In a dead system no offers can be created/updated and orders can no longer be executed. Maker can still retract their offer and withdraw their provision.

Global parameters, can update the following parameters of the system:

- Set the `Governance` address
- Set `Vault` address
- Set `Monitor` address
- Set the boolean `UseOracle`
- Set the boolean `Notify`
- Setting general parameters like updating the `Governance` address, changing the `Monitor` address

Monitor: External system contract. Used while loading the configuration when `useOracle` is active. Returns information about the `density` and `gasPrice` for the `base` / `quote` asset combination. Fully trusted that this information is correct and does not exceed `uint16` for `gasPrice` or `uint32` for `density` respectively. Additionally this contract may be called during the execution for `notifySuccess` or `notifyFail` if the notification is enabled. The contract is fully trusted to behave honestly.

Vault: Address receiving the fees

3.2 Changes in Version 3

The following functional changes were introduced in version 3:

- The separate function for single-snipes has been removed. `Snipes()` now loops over the offers one by one instead of recursively executing them. This isolates the execution of multiple snipes from each other.
- `overhead_gasbase` has been removed.
- `getConfig()` has been removed, a similar function `configInfo()` has been added.
- Events `OrderStart` and `OrderComplete` before/after market and snipe execution have been added. The field `prev` has been added to event `OfferWrite`.
- The governance address can now be specified as parameter during deployment.
- The following renaming took place:
- `base` -> `outbound_tkn`, `quote` -> `inbound_tkn`
- `statusCode` -> `mgvData`
- `restrictedCall` -> `controlledCall`

- The interface changed: Makers are now expected not to return anything upon success. Nonzero return data in the first 32 bytes will be treated as "Maker abort". This implies that for offers successfully executed the postHook callback passes empty data for parameter `makerData`.
- `updateOffer()` no longer has a return value. `retractOffer()` now returns the freed provision. Functions matchOrder and Snipe now return the penalty ("bounty").
- Functions `newOffer` and `updateOffer` can be funded when called.
- Calling permit to issue an approval now emits an approval event.

4 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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

5 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 severities. 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.

6 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	1

- No Protection for Keepers Risk Accepted

Low-Severity Findings	4

- ECDSA Signature Malleability Risk Accepted
- No Minimum Value for gasreq Acknowledged
- Redundant Check in writeOffer Acknowledged
- Spamming the Offerbook Risk Accepted

6.1 No Protection for Keepers

Design Medium Version 1 Risk Accepted

Generally, keepers may just be interested in collecting the penalty of failing offers. In Mangrove however, an offer could always succeed unexpectedly due to changing on-chain conditions. In this case, a keeper/taker may have executed an offer he did not actually intended to take and which may had a bad exchange rate. Note that offers may only fail when significant amounts of tokens are flashloaned to the maker up front but the very same offer may succeed for lower amounts.

Unaware keepers may be tricked by honeypot offers (offers that appear to fail but in reality don't fail) by malicious makers.

Keepers may protect themself by wrapping their call in a smart contract and checking for the expected outcome, but the code of Mangrove itself does not offer such a feature directly.

Risk Accepted:

Giry responded: Indeed all keepers should wrap their calls in a reverting contract. This protective wrapper does not need to be inside Mangrove. We plan to provide a standard wrapper at a separate address.

6.2 ECDSA Signature Malleability

Design Low Version 1 Risk Accepted

The `permit` function utilizes the ECDSA scheme. However missing checks for the `v`, `r` and `s` arguments allow attackers to craft malleable signatures. According to Yellowpaper Appendix F, the signature is considered valid only if `v`, `r` and `s` values meet certain conditions. The `ecrecover` for invalid values will return address 0x0 and verification will fail without informative error. The OpenZeppelin's ECDSA library performs such checks and reverts with informative messages.

Risk Accepted:

Giry responded: Code changes necessary for improved error messages would go past the contract size limit.

6.3 No Minimum Value for `gasreq`

Design | Low | Version 1 | Acknowledged

Either to create a new offer or to update an existing one, the maker must provide a value for `gasreq`. In the current implementation, there is no minimum required value. The value for gasreq may even be set to 0, which means 0 gas requirements for the calls executed on the maker's side. Nevertheless, both calls are executed, the first call to `makerExecute` with all gas defined in `gasreq` and the second to `makerPosthook` with the "leftover" gas from `gasreq`. With 0 gas these low-level calls are started but immediately revert. The system could allow these calls to be skipped when the maker sets a zero / low amount for `gasreq`.

Acknowledged:

Giry responded: The gas saved by treating `0-gasreq` as a special case is not worth the added code complexity.

6.4 Redundant Check in `writeOffer`

Design | Low | Version 1 | Acknowledged

When `writeOffer` is called a check that `ofp.gives > 0` is performed. However, the check presented below is also performed and implies the same since both `density` and `gasbase` should be positive under normal circumstances.

```
ofp.gives >=
  (ofp.gasreq + $$(local_offer_gasbase("ofp.local"))) *
    $$(local_density("ofp.local")),
```

No Issue:

Giry responded: It is possible for governance to set values such that the second check does not imply the first. The first check maintains a critical invariant.

6.5 Spamming the Offerbook

Security | Low | Version 1 | Risk Accepted

An attacker may spam the offerbook with attractive offers reverting immediately upon execution. Depending on the parameters chosen by the governance for `density` and `offer_gasbase` the resulting minimum penalty paid for the failing offer may be rather low.

Notably it is sufficient to have 85 such failing offers in the offer book (offering a very low price to ensure to be on top) to cause a revert of the transaction due to an EVM stack too deep error. Hence any transaction to `marketOrder()` of this `base`/`quote` pair will revert leaving the state unchanged.

It's still possible for keepers to clean the offerbook by sniping these offers, however such offers may reinstantiate themself during the `makerPosthook()`.

Risk Accepted:

Giry responded: Any self-reinserting spam is vulnerable to draining by any keeper.

7 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	1

- Draining All Ether Provisions of Mangrove Code Corrected

High-Severity Findings	0

Medium-Severity Findings	2

- If Condition Always True Code Corrected
- Rounding Errors In Partial Filling Code Corrected

Low-Severity Findings	7

- Call in makerPosthook Fails Silently Code Corrected
- Fields of Events Not Indexed Code Corrected
- Imprecise Comment Code Corrected
- Maximize Penalty Collected Code Corrected
- Misleading Variable Names in stitchOffers Code Corrected
- Overrestrictive Check in deductSenderAllowance Code Corrected
- Repetitive Code Code Corrected

7.1 Draining All Ether Provisions of Mangrove

Security Critical Version 1 Code Corrected

Makers can retract their offer by calling `retractOffer()`. This function accepts the boolean parameter `deprovision` which allows the maker to choose to either deprovision the offer or not.

Deprovisioning an offer credits back the provision to the maker. At the same time it must be ensured that this offer is removed from the offerbook and its gasprice must be set to 0 as the offer is no longer provisioned.

Not all possible cases are handled correctly inside function `retractOffer()`. For offers that are not live (this means they have a `0` amount for `offer.gives`) the provision can be credited back to the maker without the offer's `gasprice` being set to zero.

Hence `retractOffer()` with `deprovision` set to `true` can be executed successfully repeatedly. Consequently a maker can reclaim more provision than he initially paid for the offer. This bug allows to eventually drain all Ether of Mangrove.

An offer can easily reach `offers.gives = 0` which means it is considered to not be live:

- By calling `retractOffer()` with `bool deprovision` set to `false`, `dirtyDeleteOffer()` is executed. This call sets `offer.gives` to zero however without setting `offer.gasprice` to zero due to `deprovision` being `false`.
- After the offer has been consumed by an order `offer.gives` is `0`.

Code Corrected:

The call to `dirtyDeleteOffer()` was moved out of the `isLive` scope. Hence whenever `deprovision` is set to `true` and the provision is credited back to the `Maker`, the order is deprovisioned. Calling the function repeatedly on the same offer no longer allows to drain Ether of Mangrove, the issue has been resolved.

7.2 If Condition Always True

Design Medium Version 1 Code Corrected

During the execution of function `execute` the following check is performed:

```
if (statusCode != "mgv/notExecuted") {
  dirtyDeleteOffer(
    ...
  );
}
```

However `statusCode` cannot have the value `mgv/notExecuted` at this point so the condition is always true.

Code Corrected:

The code now runs unconditionally.

7.3 Rounding Errors In Partial Filling

Design Medium Version 1 Code Corrected

A maker's order can be partially filled according to the following snippet:

```
if (mor.fillWants) {
   sor.gives = (offerWants * takerWants) / offerGives;
 } else {
   sor.wants = (offerGives * takerGives) / offerWants;
 }
```

Note that the division can yield rounding errors. The rounding errors can be as extreme as giving funds to the maker without receiving anything in return or taking from the maker without giving anything back. For example, consider the case where the maker offers 10 A for 5 B and taker wants to take only 1 A (`fillWants == true`). Then, according to the formula she has to offer 5 * 1 / 10 = 0 B.

Code corrected

Prices are now always rounded in favor for the taker to avoid any maker draining. Hence, to calculate `sor.gives` when `fillWants` is `true` the code has been changed to:

```
uint product = offerWants * takerWants;
sor.gives =
  product /
  offerGives +
    (product % offerGives == 0 ? 0 : 1);
```

7.4 Call in `makerPosthook` Fails Silently

Design | Low | Version 1 | Code Corrected

The return value `success2` in `makerPosthook()` is never handled. Thus a failed execution of the hook can go unnoticed and unlogged.

Code Corrected:

A log event is emitted on posthook revert.

7.5 Fields of Events Not Indexed

Design | Low | Version 1 | Code Corrected

No parameter of the events defined in `MgvEvents` is marked as `indexed`. Indexing fields of events, e.g. addresses, allows to search for them easily.

```
/* Mangrove adds or removes wei from `maker`'s account */
  /* * Credit event occurs when an offer is removed from the Mangrove or when the `fund` function is called*/
  event Credit(address maker, uint amount);
  /* * Debit event occurs when an offer is posted or when the `withdraw` function is called */
  event Debit(address maker, uint amount);

  /* * Mangrove reconfiguration */
  event SetActive(address base, address quote, bool value);
  event SetFee(address base, address quote, uint value);
  event SetGasbase(
    address base,
    address quote,
    uint overhead_gasbase,
    uint offer_gasbase
  );
```

Code Corrected:

The relevant arguments were indexed.

7.6 Imprecise Comment

Correctness | Low | Version 1 | Code Corrected

At the beginning of function `execute()` the code handles whether the full offer is to be consumed or only a partial amount of the offer should be taken by the order.

```
if (
  (mor.fillWants && offerGives < takerWants) ||
  (!mor.fillWants && offerWants < takerGives) ||
  offerWants == 0
) {
  sor.wants = offerGives;
  sor.gives = offerWants;
  /* If we are in neither of the above cases, then the offer will be partially consumed. */
} else {
  /* If `fillWants` is true, we give `takerWants` to the taker and adjust how much they
  give based on the offer's price. Note that we round down how much the taker will give. */
  if (mor.fillWants) {
    /* **Note**: We know statically that the offer is live (`offer.gives > 0`) since market
     orders only traverse live offers and `internalSnipes` check for offer liveness before executing. */
    sor.gives = (offerWants * takerWants) / offerGives;
    /* If `fillWants` is false, we take `takerGives` from the taker and adjust how much they get
     based on the offer's price. Note that we round down how much the taker will get.*/
  } else {
    /* **Note**: We know statically by outer `else` branch that `offerGives > 0`. */
    sor.wants = (offerGives * takerGives) / offerWants;
  }
}
```

The last comment

> **Note**: We know statically by outer *else* branch that *offerGives* > *0*.

is not entirely correct: While it holds that `offerGives` is `> 0` this is not due to being in the outer else branch but due to it having been ensured earlier that the offer is live, so `offerGives` is `>0`. Due to being in the outer else branch we know that `offerWants` is non-zero and hence we are sure there will be no division by zero which is the important consideration here.

Code Corrected:

The comment was changed to:

> **Note**: We know statically by outer *else* branch that *offerWants* > *0*.

7.7 Maximize Penalty Collected

Design | Low | Version 1 | Code Corrected

Function `snipes()` allows keepers to snipe multiple failing offers at once and thereby collect the penalty. However, when multiple offers fail within one order, the base gas fee is split amongst all failing offers. This is done in order to distribute the base fees that applies once per transaction evenly across all affected offers.

In order to maximize their profit, professional keepers may deploy their own smart contract calling `snipe()` individually on each offer without increasing their expenses significantly, in order to ensure to always collect the maximum penalty possible.

Code corrected:

The sniping mechanism has changed and, now, `snipes()` treats all snipes in isolation. With this change `overhead_gasbase` was removed and the scenario described above no longer applies.

7.8 Misleading Variable Names in `stitchOffers`

Design | Low | Version 1 | Code Corrected

In `stitchOffers` the variable names `worseId` and `betterId` are used. However, the `betterId` refers to the offer next to the offer under consideration and `worseId` refers to the previous one. This seems to contradict the naming convention holding for the offerbook. According to this convention, for a given offer, its previous offer is a better one and its next offer a worse one.

Code Corrected:

The names of the variables were swapped.

7.9 Overrestrictive Check in `deductSenderAllowance`

Design Low Version 1 Code Corrected

`deductSenderAllowance` checks if the amount used does for a trade does not exceed the allowance the use has set. However, it prevents the full allowed amount from being used since the equality is not checked.

```
require(allowed > amount, "mgv/lowAllowance");
```

Code Corrected:

> was replaced with >=.

7.10 Repetitive Code

Design Low Version 1 Code Corrected

In `postExecute` the following snippet is used right before the call to `applyPenalty()`.

```
if (gasused > gasreq) {
   gasused = gasreq;
 }
```

Later, in `applyPenalty` the same snippet is repeated as follows:

```
if ($$(offerDetail_gasreq("sor.offerDetail")) < gasused) {
  gasused = $$(offerDetail_gasreq("sor.offerDetail"));
}
```

which is redundant.

Code Corrected:

The second check in `postExecute` was removed.

8 Notes

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

8.1 Bitwords Benefits

Note Version 1

In the implementation of Mangrove, structs to be stored in storage and handled as bitwords. This is done to improve gas efficiency. However, given that equivalent native solidity structs could fit in on storage slot, the benefits from such a decision are questionable. The main downside of this decision is the extra layer of complexity, introduced in the form of solidity code preprocessing, which aims to facilitate the handling of such bitwords.

8.2 Choosing the Parameter `gasreq`

Note Version 1

When choosing the parameter `gasreq`, makers must be aware of certain things:

As described in the code, the maker may receive only `gasreq`−`63h/64` gas, where h is the overhead of (require + cost of CALL).

Nevertheless, should the call fail due to insufficient gas the maker is accountable for this and if the overall gas remaining in the transaction is sufficient, the transaction penalizes the maker and completes successfully.

The comment states:

> We let the maker pay for the overhead of checking remaining gas and making the call.

Albeit minor, the maker also pays for the overhead to handle the return data. All of this needs to be taken account when selecting the value for `gasreq`, especially in order to ensure enough gas will be available to the call to `makerPosthook`.

8.3 Estimation of the Gas Limit for a `marketOrder` Transaction

Note Version 1

Estimating the gas limit for a transaction to `marketOrder` is tricky. Underestimating it leads to the transaction to revert, while overestimating it increases the risk of a high tx fee for a failing transaction.

Function `marketOrder` is dependent on the actual status of the offer book which may change significantly between when the transaction is generated and signed and when it is executed by being included inside a block. Offers may be added or removed resulting the gas requirement for the offers being executed as part of the order may change significantly.

While `marketOrder` can skip failing offers and refund the taker, it can only do so successfully when the transaction has enough gas.

To avoid failing transactions users have to overestimate the gas required.

8.4 Payable Fallback Functions For Taker Contracts

Note Version 1

Takers may be contracts. When makers are penalised, provision is sent to the takers by a low level call `msg.sender.call{value: amount}("")`. In this case, taker contracts must be able to handle the ether received by implementing fallback functions.

8.5 Tokens With Transfer Fees

Note Version 1

Mangrove is supposed to handle the exchange of ERC20 tokens. As shown in the snippet below, the system expects to send to the maker the same amount (`sor.gives`) it received from the taker. However, in the case of the tokens with transfer fees this trade will fail since the amount received and forwarded by Mangrove will be different than the one requested due to the fees. By providing additional balance of this token to the contract ahead of the transaction, a party may make the transfer to succeed nevertheless. This may be done by either the maker or the taker. The other party then receives less tokens then expected, as the transfer fee will be deducted.

```
if (transferTokenFrom(sor.quote, taker, address(this), sor.gives)) {
   if (
     transferToken(
       sor.quote,
       $$(offerDetail_maker("sor.offerDetail")),
       sor.gives
     )
   ) {

   ...
```