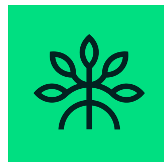
PUBLIC

Code Assessment

of the Mangrove Core Smart Contracts

November 14th, 2023

Produced for



by

CHAINSECURITY

Contents

1 **Executive Summary** 3
2 **Assessment Overview** 5
3 **Limitations and use of report** 10
4 **Terminology** 11
5 **Findings** 12
6 **Resolved Findings** 13
7 **Informational** 15
8 **Notes** 16

1 Executive Summary

Dear Mangrove team,

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

Mangrove Association (ADDMA) implements an order book-based exchange where makers can post offers that are essentially promises to trade a certain token pair for a specified amount. Takers can take these offers. When a taker takes an offer, the maker's smart contract is called and needs to fulfill the promise to exchange the tokens. If the maker does not meet their obligation, a pre-defined gas reimbursement will be given to the taker. Makers need to deposit the funds to reimburse takers when creating the offer. The project allows participants full control over their funds up until they can really be exchanged. Hence, avoiding idle or stale funds waiting for order execution.

This version implements a new internal data structure, using a tree of bitmaps in order to efficiently find the next-best offer in the order book.

Even though the codebase is complex, we did not find any severe issues. The code quality is good and Mangrove provides a good documentation for their project.

The general subjects covered are functional correctness, security and documentation. 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.

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

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 Mangrove Core 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	10 October 2023	379a0af795a388c2082bb54fd6ef7678157f0316	Initial Version
2	11 November 2023	596ed77be48838b10364b7eda1a4f4a4970c0cad	After fixes

For the solidity smart contracts, the compiler version `^0.8.10;` is specified. For the compilation version `0.8.20` is defined in the `foundry.toml`.

2.1.1 Excluded from scope

This assessment was focused on the core part of the Mangrove system. Excluded are all files outside of the `src/core` and `lib/core` directory.

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.

For a detailed system overview of Mangrove core functionality, refer to the previous audit report covering Mangrove core. The following overview will focus on the parts that changed in the updated codebase. The most significant changes include the change of the internal data structure, as well as changing the Offer interface to be defined by a tick (price) and an amount, rather than an in- and out-amount.

Mangrove Association (ADDMA) offers Mangrove Core, a decentralized exchange platform where makers can place orders that can callback to the maker and execute arbitrary code to fulfill themselves. In Mangrove Core, market makers can create offers, which are a promise that a specific smart contract will trade inbound tokens for outbound tokens at a certain price and up to a certain volume if executed. This enables market takers to trade against those orders, consuming them. In order to protect takers, makers are required to post an amount of the gas currency into Mangrove Core, so that takers can be compensated if they execute a faulty offer. Cleaners may also interact with Mangrove Core by triggering offers that they know based on simulation will not succeed. By doing so, they will collect the gas security deposit posted by the makers.

A market is specified by two tokens (A to B, but B to A would be another market) and their tick space. For each market, an offer list exists. E.g., an offer that would buy token A for token B would be in another list/market than an offer that buys token B for token A. Governance is required to define global parameters and add lists with specific local (per-list) parameters. Offers in the same tick are kept in FIFO order in a linked list called a bin. A tree structure is used to scan and retrieve bins efficiently.

To save gas and create a discretized order book, the price space is divided into discrete ticks in a fashion inspired by Uniswap V3. More precisely, for each offer list, the parameter `tickSpacing` can be set. It is at least one, then tick t corresponds to price $1.0001^{t \cdot \text{tickSpacing}}$.

Once an offer is executed successfully, even partially, it is disabled but remains provisioned with the gas currency. Makers can thus reuse the storage space allocated to the offer.

The tree structure sitting atop the bins is used to efficiently find the best (or next best) offer in an offer list. Each non-leaf node stores a presence bit for each of its children, which is set if there exists an offer in the subset of bins covered by the child. This allows the taker to quickly scan the tree and find the best price. Each leaf of the tree contains 4 bins, represented as a pair of offer ids indicating the first and last offer in the linked list. The root of the tree has 2 level1 children, each having 64 level2 children, which each has 64 level3 children, which each has 64 leaf children. This means that the tree can cover 2^{21} bins, which is enough to cover the entire price space at any tick spacing.

The index of a bin is represented as a 21-bit signed integer in two's complement representation, which can be efficiently destructured using bitwise operations:

s	aaaaaa	bbbbbb	cccccc	dd

The sign bit `s` selects the level1 node. `a`, `b`, `c` and `d` select the level2, level3, and leaf, respectively. `d` selects the bin within the leaf.

Leafs store their zeroth bin in the most significant bits, whereas level3s, level2s, and level1s store their zeroth bin in the least significant bits. In the root, the negative subtree is stored in the less significant bit.

To save gas, the local configuration of an offer list contains a cache of the nodes on the path to the current best offer. This takes up 193 bits in the same storage slot.

2.2.1 Mangrove

The `Mangrove` singleton contract contains all of the on-chain functionality of the Mangrove decentralized exchange. It is split into multiple components using inheritance. For technical reasons, the code for the `MgvView` and `MgvGovernable` components is stored in the `MgvAppendix` logic contract and accessed using `delegatecall()`.

2.2.2 MgvOfferMaking

Functions used by market makers are implemented in this component. They are as follows:

- `fund()` allows a market maker to deposit an amount in the gas currency.
- `withdraw()` allows a market maker to take back the gas currency that they deposited and that is not currently locked or forfeited.
- `newOfferByTick()` and `newOfferByVolume()` allow a market maker to post a new offer.
- `updateOfferByTick()` and `updateOfferByVolume()` allow a market maker to alter an existing offer.
- `retractOffer()` allows a market maker to cancel an existing offer. The offer can be reactivated and reused later. The offer can stay provisioned with gas currency, or the gas currency can be internally credited to the maker.
- the default function, referred to as `receive()` in solidity, is an alias for `fund()`

2.2.3 MgvOfferTaking

Functions used by market takers are implemented in this component. They are as follows:

- `marketOrderByTick()`, `marketOrderByTickCustom()`, and `marketOrderByVolume()` allow a taker to perform a market order, taking out 1 or more offers. The order may fail or partially fill, and the taker may receive reimbursement for wasted gas from failing offers.

- `cleanByImpersonation()` allows a cleaner to remove a failing offer. The cleaner must supply the address of a holder of the inbound token who set an ERC-20 approval for the Mangrove contract to proceed, but the token will never actually be spent. The cleaner will collect part of the gas currency provisioned for the offer as a bounty.

2.2.4 MgvOfferTakingWithPermit

Functions used for executing orders on behalf of market takers using allowance are implemented in this component. They are as follows:

- `marketOrderForByTick()` and `marketOrderForByVolume()` allow an address with the appropriate allowance to perform a market order on behalf of a taker.
- `approve()` allows the spender to trade on behalf of the owner. It is not to be confused with the common ERC-20 function with the same name but different arguments. The approval specifies two tokens, inbound and outbound, and an allowance denominated in the inbound token. As a result, the spender can execute market orders using the funds of the owner, and the outbound token amount is given to the owner. No explicit support exists for ERC-20 style "infinite" allowance.
- `allowance()` allows querying the trading allowance of a spender on a trading pair. It is not to be confused with the common ERC-20 function with the same name but different arguments.
- `permit()` and `DOMAIN_SEPARATOR()` allow an EOA to provide allowance to a spender using an EIP-712 signature. They are not to be confused with the common ERC-2612 functions with the same name.

2.2.5 MgvGovernance

Functions used by governance are implemented in this component. They are as follows:

- `activate()` allows governance to enable a pair of assets to be traded and set the fee, density and gas base parameters.
- `deactivate()` allows governance to disable trading for a pair of assets. It can later be reenabled with `activate()`.
- `kill()` allows governance to irrevocably kill the Mangrove contract. makers can still retract their offers and withdraw all of their gas currency.
- `setDensity96X32()` allows governance to set the minimum density parameter for offers, which is the amount of outbound token offered by offers per unit of gas consumed. If an oracle is configured and returns an in-range value, this value is ignored.
- `setFee()` allows governance to set the protocol fee charged in outbound tokens. The maximum fee is 2.56%.
- `setGasbase()` allows governance to set the per-offer list gas base parameter, which represents the amount of kilo-gas that must be set aside for Mangrove itself to process an offer internally.
- `setGasmax()` allows governance to set the global gas max parameter, which is the maximum amount of gas that an offer can provision.
- `setMaxRecursionDepth()` allows governance to set the maximum recursion depth parameter, which is the most amount of offers that can be filled by a single market order.
- `setMaxGasreqForFailingOffers()` allows governance to set the gasreq for failing offers, which bounds the amount of gas that failing offers can consume in a single market order. If this limit is exceeded, the order may be partially filled.
- `setGasprice()` allows governance to set the default gasprice in Mwei. If an oracle is configured and returns an in-range value, this value is ignored.
- `setGovernance()` allows governance to immediately transfer governance privileges to another address.

- `setMonitor()` allows governance to set the address for the monitor contract, which also serves as a gasprice and density oracle.
- `setNotify()` allows governance to toggle the notify parameter, which controls whether the monitor/oracle is notified when an offer is taken.
- `setUseOracle()` allows governance to toggle the use oracle parameter, which controls whether the oracle is used to recover gas price and density values.
- `withdrawERC20()` allows governance to recover any ERC-20 tokens held by Mangrove, whether by accident or as a result of the protocol fee.

2.2.6 MgvView

View functions are implemented in this component. Note that all functions except for `balanceOf()` and `global()` will revert if the offerlist being queried is locked. This is to avoid Makers having information about the orderbook state during `makerExecute()`, which could allow them to distinguish between "real" order execution and cleaning.

They are as follows:

- `balanceOf()` tracks the internal balance in the gas currency owned by a given maker. It is not to be confused with the common ERC-20 function with the same signature but different semantics.
- `allowance()` allows querying the trading allowance of a spender on a trading pair. It is not to be confused with the common ERC-20 function with the same name but different arguments.
- `global()` allows to query the global configuration parameters of Mangrove, packed in 256 bits
- `local()` allows to query the configuration parameters of an offer list, packed in 256 bits
- `config()` combines `global()` and `local()`
- `locked()` allows to query whether the reentrancy lock of a certain offer list is set.
- `best()` allows one to query the best-priced offer in an offer list.
- `olKeys()` allows to query the pre-image of an offer list key hash.
- `offers()` allows to query a single offer, packed in 256 bits
- `offerDetails()` allows to query details of a single offer, packed in 256 bits
- `offerData()` combines `offers()` and `offerDetails()`
- `governance()` allows to query the current governance address.
- `leafs()` allows querying a leaf of the tree structure
- `level3s()` allows to query a level 3 node of the tree structure
- `level2s()` allows to query a level 2 node of the tree structure
- `level1s()` allows to query a level 1 node of the tree structure.
- `root()` allows to query the root node of the tree structure.

2.3 Roles and Trust Model

- The contract must have a fully trusted Governance role that maintains the markets and market specifications.
- ERC20 Tokens with the following characteristics should not be used in the system: Tokens with multiple entry points, tokens with fees-on-transfer, and tokens that revert when transferring amount 0.

- We assume that the no hash collision can be found for a market's hash (`olKey.hash()`) in such a form that it would collide with another object in storage.

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:

- 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.

Severity	Count
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	0
Medium-Severity Findings	0
Low-Severity Findings	3

- Incorrect Comments Specification Changed
- Redundant Input Validation Code Corrected
- Withdraw Does Not Revert Code Corrected

Informational Findings	1

- Typo in Error String Code Corrected

6.1 Incorrect Comments

Design Low Version 1 Specification Changed

CS-MGVC-004

1. in TickTreeLib, the comment for `bestNonEmptyBinPos` is incorrect and should read:

`pos` is initially 1 if `leaf` has some nonzero bit in its MSB half, 0 otherwise. Then `pos` is `A | B`, where `A` is `iszero(pos)<<1`, so `A` is 0 if `leaf` has some nonzero bit in its MSB half, 2 otherwise. And `B` is 0 if `leaf >> (pos << 7)` has some nonzero bit in its most significant 192 bits, 1 otherwise.

2. The comment for `Bitlib.fls` indicates:

The `fls` function below is general-purpose and used by tests but not by Mangrove itself.

However, the function `DensityLib.from96X32` uses it.

3. In MgvOfferTaking, there is a typo (be instead of by):

We start be enabling the reentrancy lock for this offer list.

4. In MgvOfferTaking, the following comment above `applyPenalty()` is misleading:

If the transaction was a success, we entirely refund the maker and send nothing to the taker.

The `applyPenalty` function is never called if the transaction is successful. In this case, the provision stays in the order and the Maker is not automatically refunded.

Specification changed:

All comments have been corrected.

6.2 Redundant Input Validation

Correctness **Low** **Version 1** **Code Corrected**

CS-MGVC-005

In `marketOrderForByVolume()`, the following checks are performed:

```
require(uint160(takerWants) == takerWants, "mgv/mOrder/takerWants/160bits");
require(uint160(takerGives) == takerGives, "mgv/mOrder/takerGives/160bits");
```

These checks are redundant relative to the stricter checks performed by `ratioFromVolumes()`.

Code corrected:

The redundant checks have been removed.

6.3 Withdraw Does Not Revert

Design **Low** **Version 1** **Code Corrected**

CS-MGVC-003

When calling `withdraw()` to withdraw native tokens that were used for gas provisions, a low-level call is used. If the call fails, `withdraw()` returns with a `false` boolean as return value. It does not revert.

The `amount` is debited from the user's internal balance, so the user will not be able to withdraw anymore.

This locks the native tokens in the contract.

Code corrected:

The `withdraw` function now reverts when the low-level transfer fails and always returns true if it does not revert.

6.4 Typo in Error String

Informational **Version 1** **Code Corrected**

CS-MGVC-002

In `src/core/MgvOfferTaking.sol` line 573, the error string should be `mgv/totalGave/overflow` instead of `mgv/totalGot/overflow`.

Code corrected:

The typo has been corrected.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Permit Does Not Specify Price and Lasts Forever

Informational | Version 1 | Acknowledged

CS-MGVC-001

In MgvOfferTakingWithPermit, an allowance can be given to another address, allowing it to make orders on an address's behalf.

This allowance lets the spender take orders at any price. No `maxTick` is specified. Essentially, it is an allowance to execute a market order at any price. The impact is limited, as Mangrove only allows taking the best order on the order book. However, if orders are removed from the book inbetween the allowance being given and the order execution, the trade may have a much worse execution price than expected.

The `_allowance` given lasts forever. The spender could execute `marketOrderFor` at a much later time, when the price is totally different.

The usage here is different than in Uniswap, from which the `permit()` function was adapted. In Uniswap, the permit is used instead of an ERC20 approval. The user will still need to call the `swap()` function themselves. Here, the `_allowance` is much more powerful. It allows a trade to happen without the user making a transaction at all.

As a result, allowances should only be given by users if the full allowance is expected to be used within a short time. Otherwise, trades may be executed at unintended prices. Unused or partially used allowances should be retracted.

Acknowledged:

Mangrove Association (ADDMA) clarified that this is intended behavior:

```
This is on purpose. Additional restrictions can be set up by the authorized
contract itself, but the main purpose is to have a general delegation
mechanism that works e.g. for cold wallets that want to allow the hot wallet
some uses (such as trading on Mangrove) but not arbitrary token transfers.
```

8 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.

8.1 Gas Price Set at Offer Creation

Note **Version 1**

The `gasprice` for an order's gas provision is set at offer creation time and not updated automatically.

As a result, orders that were created at a time when gas was cheap may not have enough provision to fully reimburse the Taker if executed when the chain's gas price is higher. There will also be no incentive to clean failing orders where this is the case.

8.2 Gasmax Must Be Smaller Than Block Gas Limit

Note **Version 1**

The global `gasmax` parameter can be set by governance.

The maximum possible value for `gasmax` is `2^24 - 1`. If the value chosen is close to the block gas limit of the chain on which the contracts are deployed, a Maker will be able to create a failing order with a `gasreq` of `gasmax`. It will be impossible to clean this order, as it will be impossible to supply sufficient gas for the Maker call. The order will always revert and blame the Taker.

Note that the maximum possible value for `gasmax` is around 16.7 million, which is more than Ethereum's average gas target per block. Other chains may have lower gas block limits.

Governance should be careful not to set `gasmax` to a value that is too high, as it may cause a permanent DoS.

8.3 Large Offers Could Have Low Effective Density

Note **Version 1**

The minimum offer density (gas per token given) is enforced based on the total size (gives) of the order:

```
require(
    ofp.gives >= ofp.local.density().multiply(ofp.gasreq + ofp.local.offer_gasbase()),
    "mgv/writeOffer/density/tooLow"
  );
```

This means that the `gasreq` can be very high, as long as the total order size is very large. If an order with a very large size is created, it is likely that most Takers will only partially fill the order.

This would lead to the effective density (gas per token received) being low.

Governance can limit the maximum `gasreq` that a Maker can set by changing the global `gasmax` parameter.

8.4 Maker Can Exclude by Tx.Origin

Note Version 1

A Maker could create an order that checks `tx.origin` and fails for certain addresses.

Such an order would be impossible for another address to clean using `cleanByImpersonation()`, as the `tx.origin` cannot be impersonated.

However, the targeted address can clean the order themselves, or just accept the gas refund when taking the order.

8.5 Tokens That Revert on Zero-Transfer Not Supported

Note Version 1

ERC20-like tokens that revert when `transfer()` is called with `amount == 0` are not supported. If such a token was used, any Offer that gives that token could be cleaned by specifying `takerWants == 0`, which would incorrectly blame the Maker.

Reverting on zero-transfer does not correctly comply with the ERC20 standard. However, there are tokens in use that have this functionality. Governance should check the token's ERC20-compliance before activating a market using it.