PUBLIC

Security Audit

of Mai Protocol Smart Contracts

June 5, 2020

Produced for

Mai Protocol

by

CHAINSECURITY

Table Of Contents

Foreword 1

Executive Summary 1

Audit Overview 2

1. Methodology 2
2. Scope 2
3. Depth 3
4. Terminology 3
5. Limitations 4

System Overview 5

1. System Roles 5
2. Trust Model 6

Best Practices in MAI PROTOCOL's project 7

1. Hard Requirements 7
2. Best Practices 7
3. Smart Contract Test Suite 7

Security Issues 8

1. Floating pragma L ✓ Fixed 8
2. Unbounded loop L 8
3. `approve` not using `SafeERC20` M ✓ Fixed 8
4. Avoid using experimental features in production code L 9
5. Missing `ecrecover` check address zero L ✓ Fixed 9

Trust Issues 10

1. Whitelisted address could steal tokens M ✓ Fixed 10

Design Issues 11
1. Missing checks `LibWhitelist` M ✓ Fixed 11
2. `keccak256` hash used in constants M ✓ Fixed 11
3. `joinIncentiveSystem/exitIncentiveSystem` missing checks M ✓ Fixed 11
4. Use enums instead of casting to `uint8` L ✓ Acknowledged 11
5. Unnecessary return statement `getOrderHash` L ✓ Fixed 12
6. `matchMarketContractOrders` could be `external` M ✓ Fixed 12
7. Repeated external calls L ✓ Fixed 12
8. `LibOrder` loose checks L ✓ Fixed 12
9. `posFilledAmounts` missing check length L ✓ Fixed 12
10. Missing signature checks in `LibSignature` L ✓ Fixed 13
11. Unused constants `LONG/SHORT` M ✓ Fixed 13
12. Unused argument `mintPositionTokens` L ✓ Acknowledged 13
13. Unnecessary use of `safeTransferFrom` inside `MintingPool` M ✓ Fixed 14
14. Unnecessary use of assembly M ✓ Fixed 14
15. Unnecessary return `filledAmount` L ✓ Fixed 14
Recommendations / Suggestions 15
Addendum and General Considerations 18
1. Optional chain id and contract address in `DOMAIN_SEPARATOR` 18
2. Dependence on block time information 18
3. Outdated compiler version 18

4. Forcing ETH into a smart contract 18
5. Rounding Errors 19
Disclaimer **20**

Foreword

We would like to thank Mai Protocol for choosing ChainSecurity to audit their smart contracts. This document outlines our methodology, limitations and results.

– ChainSecurity

Executive Summary

Mai Protocol engaged ChainSecurity to perform a security audit of Mai Protocol, an Ethereum-based smart contract system. The Mai Protocol smart contracts implement a decentralized derivatives trading platform. The collateral for each derivative is bound to a single ERC20 token. Traders can exchange their collateral tokens for an equal amount of long and short position tokens. Also, traders can sell/buy their long/short position tokens from/to other traders through orders. These orders are matched on-chain using the implemented matching engine.

ChainSecurity audited the smart contracts which are going to be deployed on the public Ethereum chain. Audits of ChainSecurity use state-of-the-art tools for detection of generic vulnerabilities and checks of custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure the highest security standards.

During the audit ChainSecurity did not discover any high or critical severity issues. However, 21 security, trust and design issues of medium and low severity were found. Mai Protocol has fixed all reported medium severity issues, and acknowledged or fixed most of the low severity issues and suggestions. Two low severity security issues have not been resolved and remain open.

The newest code we received on June 1st, 2020, does contain fixes to our issues and new unrelated changes. We did not assess the security of the unrelated changes.

Audit Overview

Methodology

ChainSecurity's methodology in performing the security audit consisted of four chronologically executed phases:

1. Understanding the existing documentation, purpose and specifications of the smart contracts.
2. Executing automated tools to scan for generic security vulnerabilities.
3. Manual analysis covering both functional (best effort based on the provided documentation) and security aspects of the smart contracts by one of our ChainSecurity experts.
4. Preparing the report with the individual vulnerability findings and potential exploits.

Scope

Source code files received	September 25, 2019
First git commit	`4ef3682c594a2a836ccd9abaebacbacaff853183`
EVM version	Byzantium
Initial Compiler	solc compiler, version 0.5.2
Updated source code files received	June 1, 2020
Second git commit	`325a9d2ef5979f70ebfceef3fbd1f18d20514d5c`
Updated Compiler	solc compiler, version 0.5.8

The scope of the audit is limited to the following source code files.

In Scope	File	SHA-256 checksum
✓	MaiProtocol.sol	`dfc963aa59aaa31ec7179ee1e11299faafad4b1a297302ad5d3da7836f9895d9`
✓	MintingPool.sol	`a8bba19a6884e87d4085a97298db652a34e6793925b5ce28b8511062fa642416`
✓	Proxy.sol (file removed in code version with fixes)	`2cc9c684a29e44b5f6f5fb4d3ba2caa86b773f0ae795590b0616e327ab538199`
✓	interfaces/IMarketCollateralPool.sol	`36c686685a5958d7b42f9748afb8438379876db7f30b63045e673625e79aa357`
✓	interfaces/IMarketContract.sol	`7ce73360d7194178e0c145107d2947e55fc67b574c4eecfed80080f65c8921e3`
✓	interfaces/IMarketContractRegistry.sol	`0322f96b4ec0b934726c1847bd9a37c02731e24e2ce26092182eb772d4060e37`
✓	lib/EIP712.sol	`b33301ed37e564671cebbaf7d38c1dd24161b1cffcec8148fa3fc465c1fad6fc`
✓	lib/LibExchangeErrors.sol	`aed1ea2b997f5f7273cee15b22acbadd8da495201d1a03e777dd4f2ff0cbd5ce`
✓	lib/LibMath.sol	`42dc53dd0adea9332df2983a6cc1c4a3bbe3b748a119221e7a6d8d9bbeeb446b`
✓	lib/LibOrder.sol	`25a29428491a2643ee475ee9815c3c7390c8a8c97820a063fe8f0b6df731bf61`
✓	lib/LibOwnable.sol	`39e7d7299019f6effa6f1f3d016daaf9cb528dc01927958046281f1257a7d64b`
✓	lib/LibRelayer.sol	`e12da946f5e24002c16c10172bb51b7457679fb0047fec345abcefa387ff0e8e`
✓	lib/LibSignature.sol	`024dc554ff40f034482f54d9ac64d6223cdde9958865a7cbdf03717cbf748f51`
✓	lib/LibWhitelist.sol	`c9570e2b6407eaa462d6e4388670f25aefabe2b3a9202dd63166f60d38c74eeb`

For these files the following categories of issues were considered:

In Scope	Issue Category	Description
✓	Security Issues	Code vulnerabilities exploitable by malicious transactions
✓	Trust Issues	Potential issues due to actors with excessive rights to critical functions
✓	Design Issues	Implementation and design choices that do not conform to best practices

Depth

The security audit conducted by ChainSecurity was restricted to:

- Scanning the contracts listed above for generic security issues using automated systems and manually inspecting the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, ChainSecurity has adopted the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business-related consequences of an exploit.

Severity is derived from the likelihood and the impact calculated previously.

We categorise the findings, depending on their severities, into four distinct groups:

- L Low: can be considered less important
- M Medium: should be fixed
- H High: we strongly recommend fixing it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the table below, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit, concerns might arise or tools might flag certain security issues. After carefully inspecting the potential security impact, we assign the following labels:

- ✓ No Issue no security impact
- ✓ Fixed the issue is addressed technically, for example by changing the source code
- ✓ Addressed the issue is mitigated non-technically, for example by improving the user documentation and specification
- ✓ Acknowledged the issue is acknowledged and it is decided to be ignored, for example due to conflicting requirements or other trade-offs in the system

Findings that are labeled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is listed simply to give the reader a quick overview of what kind of issues were found during the audit.

[1] `https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology`

Limitations

Security auditing cannot uncover all existing vulnerabilities: even a contract in which no vulnerabilities are found during the audit is not a guarantee of a secure smart contract. However, auditing enables 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 smart contract application, while others lack protection only in certain areas. This is why we carry out a source code review aimed at determining all issues that need to be fixed. Within the customer-determined timeframe, ChainSecurity has performed a security audit in order to discover as many vulnerabilities as possible.

System Overview

The MAI PROTOCOL smart contract system implements a derivatives trading platform. Derivatives are implemented in other projects, such as Market Protocol, UMA Protocol and Yield Protocol. What the MAI PROTOCOL project offers is a trading platform to trade derivatives created in these other projects. This initial version only supports the Market Protocol[2] derivatives.

Each derivative is bound to one underlying collateral ERC20 token, for example DAI. Each derivative also has two position ERC20 tokens named long and short. The ERC20 long/short position tokens of derivatives can be send directly to other accounts. However, there is no trading platform which allows adding orders for derviatives to be added, which will then be matched together and atomically traded. If for example you want to go long you want to either sell some of your short position tokens to another user, or buy long position tokens from another user.

The order-matching engine implemented in the `MaiProtocol` contract will try to match multiple orders. An order can be resolved in four ways:

- selling position tokens for collateral
- buying position tokens with collateral
- minting position tokens for collateral
- redeeming position tokens for collateral

The order matching engine only accepts off-chain pre-signed orders. Each order has a relayer parameter (part of the signed data). Orders are matched by calling `matchMarketContractOrders`. This function can only be called by the relayer of the orders. Therefore, all of the orders in a single call need to have the same relayer. A relayer can also register other addresses to act as delegate (sub-relayers).

Apart from the `MaiProtcool` contract, MAI PROTOCOL has a so called `MintingPool` contract. Using this contract is optional. If there is no `MintingPool`, the `MaiProtocol` contract will instead directly call the Market Protocol contract. The `MintingPool` contract can act as a buffer of long/short position and collateral tokens. This buffer makes calling the Market Protocol contract unnecessary when the `MintingPool` token balance is sufficient.

System Roles

This section outlines the different roles' permissions and purposes within the system.

MintingPool The `MintingPool` contract is optional. If there is no `MintingPool`, the `MaiProtocol` contract will instead directly call the Market Protocol contract.

Owner The owner can call the following functions:

- `approveERC20`, to approve a spender to transfer any ERC20 token of this contract.
- `withdrawERC20`, to withdraw any ERC20 token belonging to this contract.
- `internalMintPositionTokens`, converting collateral in pool to position tokens for further minting requests.
- `internalRedeemPositionTokens`, converting position tokens in pool to collateral tokens for further redeeming requests.

Whitelisted The `MintingPool` will add the `MaiProtcool` contract to the whitelist after deployment. A whitelisted address can call the following functions:

- `mintPositionTokens`, mint position tokens from collateral pool, then send minted tokens to caller.
- `redeemPositionTokens`, redeem position tokens from collateral pool, then send redeemed tokens to caller.

MaiProtocol Called MaiExchange in the documentation.

Owner The owner can call the following functions:

- `approveERC20`, to approve a spender to transfer any ERC20 token of this contract.
- `withdrawERC20`, to withdraw any ERC20 token belonging to this contract.

[2] marketprotocol.io

- `setMarketRegistryAddress`, set the address of the MarketRegistry contract.
- `setMintingPool`, set the address of the Mai `MintingPool` contract.

Relayer Each order has a `relayer` parameter. This address is allowed to match the orders by calling `matchMarketOrders`. A relayer can call:

- `matchMarketContractOrders`, match taker and maker orders and settle the results.
- `cancelOrder`, mark an order as cancelled.

Anybody Anybody can call the following functions:

- `cancelOrder`, mark an order as cancelled.

Trust Model

Here, we present the trust assumptions for the roles in the system as provided by MAI PROTOCOL. Auditing the enforcement of these assumptions is outside the scope of the audit. Users of MAI PROTOCOL should keep in mind that they have to rely on MAI PROTOCOL to correctly implement and enforce these trust assumptions.

Deployer The deployer is *trusted* to use the correct code during deployment and set the right parameters.

Owner The owner of each contract is *trusted* to call the right functions with valid parameters.

Relayer A relayer is *semi-trusted*. Initially MAI PROTOCOL is expected to offer a (trusted) relayer to users. However, anybody can become an (untrusted) relayer. Therefore, relayers are assumed to be potentially malicious.

User A regular user is *untrusted* and assumed to be potentially malicious.

Best Practices in MAI PROTOCOL's project

CHAINSECURITY is determined to deliver the best results to ensure the security of a project. To enable us to do so, we are listing Hard Requirements which must be fulfilled to allow us to start the audit. Furthermore we are providing a list of proven best practices. Following them will make audits more meaningful by allowing efforts to be focused on subtle and project-specific issues rather than the fulfilment of general guidelines.

Hard Requirements

These requirements ensure that the MAI PROTOCOL's project can be audited by CHAINSECURITY.

- [x] **All files and software for the audit have been provided to CHAINSECURITY**
  The project needs to be complete. Code must be frozen and the relevant commit or files must have been sent to CHAINSECURITY. All third party code (like libraries) and third-party software (like the solidity compiler) must be exactly specified or made available. Third party code can be located in a folder separated from client code (and the separation needs to be clear) or included as dependencies. If dependencies are used, the version(s) need to be fixed.
- [x] The code must compile and the required compiler version must be specified. When using outdated versions with known issues, clear reasons for using these versions are being provided.
- [x] There are migration/deployment scripts executable by CHAINSECURITY and their use is documented.
- [x] The code is provided as a Git repository to allow reviewing of future code changes.

Best Practices

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable.

- [ ] There are no compiler warnings, or warnings are documented.
  EXPLANATION: MAI PROTOCOL uses experimental `ABIEncoderV2` pragma.
- [x] Code duplication is minimal, or justified and documented.
- [x] The output of the build process (including possible flattened files) is not committed to the Git repository.
- [x] The project only contains audit-related files, or, if this is not possible, a meaningful distinction is made between modules that have to be audited and modules that CHAINSECURITY should assume are correct and out-of-scope.
- [x] There is no dead code.
- [x] The code is well-documented.
- [x] The high-level specification is thorough and enables a quick understanding of the project without any need to look at the code.
- [x] Both the code documentation and the high-level specification are up-to-date with respect to the code version CHAINSECURITY audits.
  EXPLANATION: There are numerous mistakes in the `settleResults` comments.
- [x] Functions are grouped together according to either the Solidity guidelines[3], or to their functionality.

Smart Contract Test Suite

In this section, CHAINSECURITY comments on the smart contract test suite of MAI PROTOCOL. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

The provided tests are extensive and test both success and failure cases.

[3] `https://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions`

Security Issues

This section relates to our investigation into security issues. It is meant to highlight times when we found specific issues, but also mentions what vulnerability classes do not appear, if relevant.

Floating pragma L ✓ Fixed

MAI PROTOCOL uses a floating pragma solidity ^0.5.2. Contracts should be deployed with the same compiler version and flags that have been used during testing and the audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively[4].

Likelihood: Low
Impact: Low

Fixed: MAI PROTOCOL now uses `0.5.8`

Unbounded loop L

In the `LibWhitelist` contract the `removeAddress` function contains an unbounded loop to remove an address from the `allAddresses` array.

```
for(uint i = 0; i < allAddresses.length; i++){
    if(allAddresses[i] == adr) {
        allAddresses[i] = allAddresses[allAddresses.length - 1];
        allAddresses.length -= 1;
        break;
    }
}
```

In case the `allAddresses` array contains n addresses, the loop's worst case complexity will be $O(n)$. Hence, it might be possible that this loop would consume more gas than the block gas limit. Which would result in an out-of-gas exception.

When EIP-1884 has been implemented as part of the upcoming Istanbul hard fork, the cost of SLOAD will be increased. Therefore the loop might run out-of-gas sooner.

Although it is unlikely that the whitelist will contain so many entries to make it run out-of-gas. If it does, no more whitelisted addresses can be removed. Therefore, MAI PROTOCOL could still consider not using a loop, by for example storing the index of the address in the array in a mapping.

Likelihood: Low
Impact: Medium

`approve` not using `SafeERC20` M ✓ Fixed

The `Proxy` contract uses `SafeERC20` for all `IERC20` calls. Instead of using `transfer`, MAI PROTOCOL correctly uses `safeTransfer`. MAI PROTOCOL however forgot to use `safeApprove` instead of `approve`. Therefore, the `approve` calls are not checking the return value. The same applies to the `approve` calls inside `MintingPool`. MAI PROTOCOL should use `safeApprove` instead of `approve`.

Likelihood: Low
Impact: High

Fixed: MAI PROTOCOL removed the `Proxy` contract and updated `MintingPool` to use `safeApprove`.

[4] https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md

Avoid using experimental features in production code L

The smart contracts of MAI PROTOCOL make use of `pragma experimental ABIEncoderV2`. This is considered unsafe and should not be used in live production code. MAI PROTOCOL is recommended to not use the experimental pragma.

Likelihood: Low
Impact: Low

Missing `ecrecover` check address zero L ✓ Fixed

The `isValidSignature` function uses `ecrecover` to recover the signer address. If the signature is invalid `ecrecover` will return address zero. At the end of `isValidSignature` the recovered address is compared to the `signerAddress`, the result being returned as a boolean. If the signature is invalid, and thus address zero is recovered, and the passed in `signerAddress` is also address zero, the `isValidSignature` function will return `true`, meaning the signature was valid.

The `signerAddress` argument is originating from `orderParam.trader`, and will also be used to transfer tokens from. To transfer tokens `safeTransferFrom` is used, which requires an upfront approval being set by address zero. In most tokens there can be no approval set by address zero because nobody has its private key. Some tokens that are implemented wrong could allow setting an approval for address zero, but such tokens are rare and will likely not be used by MAI PROTOCOL.

Since it is highly unlikely that an approval has been set by address zero for any token, this issue does not lead to any direct problems. Still, MAI PROTOCOL should add a check that the recovered address from `ecrecover` does not equal address zero. For a reference implementation that includes the missing check see the OpenZeppelin `ECDSA` contract[5].

Likelihood: Low
Impact: Medium

Fixed: MAI PROTOCOL added the missing check.

[5] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/cryptography/ECDSA.sol

Trust Issues

This section reports functionality that is not enforced by the smart contract and hence correctness relies on additional trust assumptions.

Whitelisted address could steal tokens M ✓ Fixed

In the `Proxy` contract only whitelisted addresses are allowed to call `mintPositionTokens` and `redeemPositionTokens`. Both of these functions take in a `contractAddress` argument, which represents the address of the Market Contract. From looking at the tests CHAINSECURITY noticed only the `MaiProtocol` contract will be added to this whitelist. Still, it is possible for the owner to add other addresses to the whitelist. These addresses could then add a fake Market Contract, with a fake Market Contract Pool. However, it is unlikely anybody would interact with this contract.

If only the `MaiProtocol` is supposed to be added to the whitelist, MAI PROTOCOL could instead use an **`address`** or `IMaiProtocol` variable in which to store the address of the `MaiProtocol`. Thereby preventing the owner from adding another address.

Instead of letting users trust the owner to not add any other addresses, MAI PROTOCOL is advised to replace the whitelist with a single **`address`** variable.

Fixed: MAI PROTOCOL removed the `Proxy` contract.

Design Issues

This section lists general recommendations about the design and style of MAI PROTOCOL's project. These recommendations highlight possible ways for MAI PROTOCOL to improve the code further.

Missing checks `LibWhitelist` M ✓ Fixed

In the `LibWhitelist` contract the `addAddress` function can be used by the Owner to add an address to the whitelist. However, there is no check that makes sure the address is not yet whitelisted. Therefore, the same address can be added multiple times. Each time an event `AddressAdded` is emitted, and the address is pushed onto the `allAddresses` array.

When the Owner removes an address from the whitelist by calling `removeAddress`, there is no check that makes sure the address is currently on the whitelist. If an address is not on the whitelist the `removeAddress` function will still succeed and emit an `AddressRemoved` event, even though nothing was removed. If the same address is present multiple times in the whitelist, it can only be removed by calling `removeAddress` multiple times.

As explained in another issue it might make more sense to not use a whitelist. However, if MAI PROTOCOL wants to keep using `LibWhitelist` the missing checks should be added to `addAddress` and `removeAddress`.

Fixed: MAI PROTOCOL added the missing checks to `addAddress` and `removeAddress`.

`keccak256` hash used in constants M ✓ Fixed

The `LibOrder` and `EIP712` contracts define a keccak256 hash output as a constant variable. The Solidity compiler is currently unable to generate these hashes upon compilation (though it may do so in the future). Therefore, these hashing operations will be executed upon every access to this constant, resulting in unnecessary gas fees.

CHAINSECURITY recommends to either hardcode these hashes and assert them in the constructor, or generate and assign them in the constructor. For more info please see this GitHub issue[6].

Fixed: MAI PROTOCOL replaced the keccak256 calls with the hardcoded keccak256 hash.

`joinIncentiveSystem`/`exitIncentiveSystem` missing checks M ✓ Fixed

The `joinIncentiveSystem` function inside `LibRelayer` allows anybody to join/enable the incentive system. It does this by deleting the `hasExited` mapping value of `msg.sender`. After which it emits a `RelayerJoin` event. There is currently no check if the `msg.sender` already has joined the incentive system. Therefore, if an address which already joined the incentive system calls this function again, it will again emit an event `RelayerJoin`, even though no new relayer joined.

The same applies to `exitIncentiveSystem` where there is no check that the caller previously joined the incentive system.

MAI PROTOCOL should add the missing checks.

Fixed: MAI PROTOCOL replaced both functions with the functions `approveDelegate`/`revokeDelegate` and applied the missing checks to these functions.

Use enums instead of casting to `uint8` L ✓ Acknowledged

The`isValidSignature` function inside `LibSignature` contract casts the method argument `OrderSignature.config[1]` to a `uint8` after which it is compared to the enum by also casting the enum to a `uint8`. Instead of this double cast to `uint8`, the initial cast could immediately cast the `uint8` into the enum `SignatureMethod`.

[6] `https://github.com/ethereum/solidity/issues/4024`

This would make the casting of the enum inside the if check to a uint8 unnecessary and would make the code more readable.

There are numerous other places throughout the code where instead of comparing an enum, the enum is casted to a uint8. MAI PROTOCOL should not cast enums to a uint8 but instead use the enums for comparing.

Acknowledged: MAI PROTOCOL explained that: "Using unknown value for enum would cause a 'invalid opcode' error which sometimes may hide the revert message shown on etherscan. We decide to leave it unchanged."

Unnecessary return statement getOrderHash L ✓ Fixed

In the LibOrder contract the getOrderHash function has defined a return variable named orderHash.

```
function getOrderHash(Order memory order) internal view returns (bytes32
    orderHash) {
  orderHash = hashEIP712Message(hashOrder(order));
  return orderHash;
}
```

Solidity will automatically return the named return variable. Hence, there is no need to explicitly return it.

Fixed: MAI PROTOCOL removed the return statement.

matchMarketContractOrders could be external M ✓ Fixed

The matchMarketContractOrders function is currently declared as public, even though it is not called from inside the contract. Functions with visibility external can directly read from calldata and do not need to first copy function arguments to memory. Therefore, declaring this function as external will lower the gas cost when calling this function.

Fixed: MAI PROTOCOL declared the function as external.

Repeated external calls L ✓ Fixed

In the MintingPool contract the redeemPositionTokens function calls marketContract.COLLATERAL_TOKEN_ADDRESS() multiple times. Instead of calling this function multiple times, MAI PROTOCOL could store the result in a variable. By lowering the amount of function calls the gas cost of executing the withdrawCollateral function will decrease.

Fixed: MAI PROTOCOL updated the code to only make one external call, saving the result in a local variable for later use.

LibOrder loose checks L ✓ Fixed

Inside LibOrder the function isSell returns true if the sell/buy byte inside the bytes32 data equals 1. There is no isBuy function, but instead the negation of isSell is used to check if an order is a buy. A buy should have the sell/buy byte set to 0. By not explicitly checking that this byte is 0, any value besides 1 will be seen as a buy order. Although this does not lead to any problems, CHAINSECURITY thinks this is bad design and advises MAI PROTOCOL to explicitly check that the byte is 0. The same applies to isMakerOnly.

Fixed: MAI PROTOCOL updated the checks such that 0 equals buy and any non-zero value equals sell.

posFilledAmounts missing check length L ✓ Fixed

The posFilledAmounts variable contains an array of uint256. The array is meant to be of equal length to the makerOrderParams. However, no such check exists. This will lead to a revert with no descriptive message. MAI PROTOCOL should consider adding a check to make sure the posFilledAmounts length equals the makerOrderParams length, and return a descriptive error message if it does not.

Fixed: MAI PROTOCOL added the missing check.

Missing signature checks in `LibSignature` L ✓ Fixed

The signature checking code inside `LibSignature` is missing two essential checks for ECDSA signatures. First off, there is no check to make sure `v` equals `27` or `28`. Second, there is no check to make sure `s` is in the lower half.

Each of these two missing checks allow signature malleability, whereby specific different values for `v` and `s` lead to the same address being recovered. If the `v` or `s` is changed, the same address will be recovered as for the order with the original `v` and `s`.

However, the calculated `orderHash` will be the same, thereby preventing signature malleability from causing any problems. Still, MAI PROTOCOL is advised to implement the two missing signature checks. For a reference implementation of the missing checks see the OpenZeppelin `ECDSA` contract[7].

Fixed: MAI PROTOCOL added the missing checks.

Unused constants `LONG`/`SHORT` M ✓ Fixed

There are two constant variables defined in `MaiProtocol` for "long" and "short".

```
uint256 public constant LONG = 0;
uint256 public constant SHORT = 1;
```

Although some places in the code use the above constants, there are still some other places that use a literal `0` or `1`. For example:

```
// margins and balances are both a uint[2]
orderInfo.margins[0] = calculateLongMargin(orderContext, orderParam);
orderInfo.margins[1] = calculateShortMargin(orderContext, orderParam);
orderInfo.balances[0] = getERC20Balance(orderContext.posAddresses[0],
    orderParam.trader);
orderInfo.balances[1] = getERC20Balance(orderContext.posAddresses[1],
    orderParam.trader);
```

Yet another way of storing this information is the `OrderContext` struct where the `takerSide` variable is defined as:

```
uint256 takerSide; // 0 = buy/long, 1 = sell/short
```

Having all these different ways of storing/using "long" or "short" makes the code less readable. MAI PROTOCOL should instead choose one type of storing/using it and use that consistently throughout the codebase. To improve the readability of the code MAI PROTOCOL could use an enum, just like Market Protocol:

```
enum MarketSide { Long, Short}
```

MAI PROTOCOL is advised to use an enum, and to use it consistently throughout the code.

Fixed: MAI PROTOCOL now consistently uses the defined constants.

Unused argument `mintPositionTokens` L ✓ Acknowledged

The third argument of the `mintPositionTokens` function inside `MintingPool.sol` is not used. MAI PROTOCOL should consider removing it and not passing it in from `Proxy.mintPositionTokens`.

Acknowledged: MAI PROTOCOL acknowledged the issue but explained that "Since the MintingPool is a plugable component, we need to keep the signature of mint/redeem consistent with what in MPX contract. We decide to leave it unchanged."

[7] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/cryptography/ECDSA.sol

Unnecessary use of safeTransferFrom inside MintingPool M ✓ Fixed

The MintingPool uses safeTransferFrom inside redeemPositionTokens/mintPositionTokens to transfer long/short/collateral tokens from the caller (Proxy) to the MintingPool itself. Thereby requiring upfront approval by Proxy for MintingPool to transfer the tokens of Proxy. ChainSecurity thinks this design could be improved by transferring the tokens from Proxy.mintPositionTokens/Proxy.redeemPositionTokens to MintingPool before calling MintingPool.mintPositionTokens/MintingPool.redeemPositionTokens. Doing this would make the approval of Proxy to let MintingPool transfer its long/short/position tokens unnecessary. This would lower the overall complexity of the Mai Protocol smart contract system.

Fixed: Mai Protocol removed the Proxy contract.

Unnecessary use of assembly M ✓ Fixed

There are several functions inside MaiProtocol that use assembly. The use of assembly is generally not recommended unless it offers a clear benefit or is required to perform an action which Solidity does not provide. Each of the functions inside MaiProtocol that use assembly only call a single function inside Proxy. ChainSecurity does not see any reason to use assembly for this. Because of this, ChainSecurity recommends Mai Protocol to get rid of all the assembly inside MaiProtocol.sol. Mai Protocol should instead simply call the function in the proxy, e.g. IProxy(proxyAddress).transfer(token, to, value).

The use of assembly is not limited to the MaiProtocol contract. The LibOrder.hashOrder function also uses assembly to prepend a constant in a memory slot before that of the passed in struct. The trade-off between readability and gas cost savings is not worth the use of assembly to simply calculate a keccack256. Mai Protocol could instead add the constant (EIP712_ORDER_TYPE) to the struct. The struct is not stored in **storage**, but is only used in **memory**. Therefore, gas cost increase should be minimal.

Fixed: Mai Protocol removed the usage of assembly from MaiProtocol.sol

Unnecessary return filledAmount L ✓ Fixed

The fillMatchResult function returns the filled amount. However, this value is also assigned to the memory MatchResult result variable. Since this is a memory struct variable it is by reference. Therefore the updating of result.posFilledAmount is sufficient, no need for returning the filled amount.

Also, inside getMatchResult the filledAmount is already present in result.posFilledAmount.

Fixed: Mai Protocol removed the unnecessary return of the filled amount.

Recommendations / Suggestions

☐ In the `LibWhitelist` contract, the `removeAddress` function reduces the `allAddresses` array length by 1 to remove the last element from the array.

```
if(allAddresses[i] == adr) {
  allAddresses[i] = allAddresses[allAddresses.length - 1];
  allAddresses.length -= 1;
  break;
}
```

However, in Solidity `^0.5.0`, a new member function `pop()` exists for array data types. This function can be used to remove the last item of an array, and will throw when the array is already empty (preventing underflow).

☑ The `getPartialAmountFloor` function inside `LibMath` calculates a percentage of a certain value using a numerator and denominator. From looking at the tests it appears the actual order of the variables is `multiple`, `denominator`, `numerator`. It does not matter for the outcome as the `numerator` and `multiple` are only multiplied. Still, MAI PROTOCOL could update the code or the tests.

☐ The function `transferOwnership` inside `LibOwnable.sol` transfers the ownership directly to the address given as a function argument. In case of any mistake, the ownership will be transferred to some random account and is most likely "lost". Therefore, client could consider using a scheme in which the new owner needs to claim the ownership, to finally get it. This makes sure that at least the account is controlled by some user.

☑ The `transferOwnership` function checks that `newOwner` is not address zero. However, it does not check that `newOwner` differs from the current owner. MAI PROTOCOL could consider adding such a check.

☐ Inside the `LibWhitelist` contract the `allAddresses` variable is defined as an **`address`** array. There is a function `getAllAddresses` which returns the entire list. If MAI PROTOCOL would like to know the length of the whitelist there is no direct way of doing this without retrieving the entire list. Therefore, MAI PROTOCOL could add a function which only returns the length of the `allAddresses` array.

☑ In the `mai.md` documentation file under the Solution section, four different matching types are described. The fourth one is mentioned as below:

Trader A's Side	Trader A's Position	Trader B's Side	Trader B's Position	Mai Protocol Smart Contract Process
Buy	Negative	Sell	Positive or Zero	Transfer the short position token from A to B and transfer the collateral token from B to A

The above matching type is not possible. Instead, if the `Trader B's Position` is `Negative or Zero` then it would make sense.

CHAINSECURITY recommends correcting the documentation.

☑ In the `MaiProtocol` contract the `doBuy` function comment reads:

```
/**
 * doBuy: taker buy position token from maker.
 *          taker -> maker: position
 *          maker -> taker: collateral
 *          taker -> relayer: fee
 */
```

However, inside the code body, the position tokes are sent from maker —> taker and collateral tokens are sent from taker —> maker.

ChainSecurity recommends correcting the code comment.

In the mai.md documentation the matchOrders function is mentioned. However, there is no function with this name in the code.

ChainSecurity recommends correcting the function name in the documentation.

Inside the getOrderInfo function three checks are performed:

```
if (orderInfo.filledAmount >= order.amount) {
  status = uint8(OrderStatus.FULLY_FILLED);
} else if (block.timestamp >= getExpiredAtFromOrderData(order.data)) {
  status = uint8(OrderStatus.EXPIRED);
} else if (cancelled[orderInfo.orderHash]) {
  status = uint8(OrderStatus.CANCELLED);
}
require(status == uint8(OrderStatus.FILLABLE), ORDER_IS_NOT_FILLABLE);
```

Since the status is only used to check if it is valid in the **require**, this could be rewritten to three **require** statements. Thereby getting rid of the if/else and the last **require**. Mai Protocol should consider rewriting the code to use **require** instead of if/else.

In the MintingPool contract the mintPositionTokens function has a local variable:

```
uint256 neededMakretToken = calculateMarketTokenFee(marketContract,
    qtyToMint);
```

The spelling of the variable name is incorrect, it should be neededMarketToken.

There are four events defined inside MintingPool.sol. The first two have a third parameter called value. However, for the last two event names this parameter is called amount. Mai Protocol could consider normalizing the event argument names.

The function comment of doRedeem reads:

```
*  for FillAction.MINT
```

This should instead be: **for** FillAction.REDEEM.

Inside many functions in the Mai Protocol smart contracts a variable named marketContractPool is defined.

```
IMarketContractPool marketContractPool =
    IMarketContractPool(marketContract.COLLATERAL_POOL_ADDRESS());
```

Inside the Market Protocol contracts there is only one global collateral pool, which is a separate contract named MarketCollateralPool. Each of the MarketContract contracts uses this same MarketCollateralPool. Therefore, naming it marketContractPool is misleading as it is not a pool specifically for that market contract. Naming this variable marketCollateralPool better reflects what it is. Also, the interface IMarketContractPool should instead be called IMarketCollateralPool.

The comments inside the settleResult function contain numerous mistakes. For example:

```
*  ============= doRedeem =============
*  - taker   ->    proxy   : pos
*  - maker   ->    proxy   : pos-opposite
*  - proxy   ->    mpx     : redeem
*  - proxy   ->    maker   : ctk - makerFee
*  ====================================
*  - proxy   ->    taker   : ctk - makerFee
*  - proxy   ->    relayer : ctk + makerFee + takerFee
```

This should be:

```
*  ============= doRedeem =============
*  - taker   ->    proxy   : pos-opposite
*  - maker   ->    proxy   : pos
*  - proxy   ->    mpx     : redeem
*  - proxy   ->    maker   : ctk - makerFee - makerGasFee
*  ====================================
*  - proxy   ->    taker   : ctk - takerFee - takerGasFee
*  - proxy   ->    relayer : ctk + makerFee + takerFee + makerGasFee +
    takerGasFee
```

MAI PROTOCOL is advised to fix the incorrect comments.

✓ The `approveCollateralPool` function inside `Proxy.sol` allows setting an allowance for the collateral token, long position token, and short position token. The `amount` function argument defines how much to approve and is used for approval of all three tokens. MAI PROTOCOL could add two more function arguments to make it possible to set approval per token. The same applies to `approveCollateralPool` in `MintingPool`.

Addendum and General Considerations

Blockchains and especially the Ethereum Blockchain might often behave differently from common software. There are many pitfalls which apply to all smart contracts on the Ethereum blockchain.

In this section, ChainSecurity mentions general issues which are relevant to Mai Protocol's code, but do not require a fix. Additionally, in this section ChainSecurity clarifies or extends the information from the previous sections of this security report. This section, therefore, serves as a reminder to Mai Protocol and to raise awareness of these issues among potential users.

Optional chain id and contract address in `DOMAIN_SEPARATOR`

There is currently no chain id present in the `DOMAIN_SEPARATOR`. This means a testnet signature can be replayed on the mainnet. To make this work all of the params used to create the `orderHash` should be the same on both the testnet and the mainnet. This includes the `marketContractAddress`, which is highly unlikely. Still, Mai Protocol could add a chain id in the domain separator.

There is another piece of information which could be added to the `DOMAIN_SEPARATOR`, the `MaiProtocol` contract address. If this piece of information is not part of the `DOMAIN_SEPARATOR`, and Mai Protocol deploys a new `MaiProtocol` contract in the future because the old for example contains a bug, than the orders for the original `MaiProtocol` contract would also be accepted by the new `MaiProtocol` contract. If on the other hand the `MaiProtocol` address is part of the `DOMAIN_SEPARATOR`, and this address is checked to be equal to the current contract's address, than orders for the original `MaiProtocol` would not be accepted by any new `MaiProtocol` contract as the contract address would not match. The orders would have to be recreated for the new `MaiProtocol` contract and signed by the users. Not having the `MaiProtocol` contract address inside the `DOMAIN_SEPARATOR` can thus be seen as an issue or a feature to allow seamless upgrading to new `MaiProtocol` contracts.

For a more in-depth explanation of these two (and more) optional EIP712 fields see this article by MetaMask[8].

Dependence on block time information

Mai Protocol uses `block.timestamp` / `now` inside the `MaiProtocol` contract. Although block time manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by around 15 seconds compared to the actual time. However, in the context of the project and given the required effort, this is not perceived as an issue[9].

Outdated compiler version

ChainSecurity could not find obvious issues with the compiler version Mai Protocol is using. Mai Protocol uses solc compiler, version 0.5.8. If Mai Protocol is aware of the compiler's behavior and bugs, there might be good reasons for using an older compiler version. While the latest version does contain bug fixes, it might introduce new bugs.

ChainSecurity does, however, recommend to use the same compiler version homogeneously throughout the project and to use the compiler version for deployment that was used during testing. Furthermore, for any used version it is helpful to monitor the list of known bugs[10].

Forcing ETH into a smart contract

Regular ETH transfers to smart contracts can be blocked by those smart contracts. On the high-level this happens if the according solidity function is not marked as `payable`. However, on the EVM levels there exist different techniques to transfer ETH in unblockable ways, e.g. through `selfdestruct` in another contracts. Therefore, many contracts might theoretically observe "locked ETH", meaning that ETH cannot leave the smart contract any more. In most of these cases, it provides no advantage to the attacker and is therefore not classified as an issue.

[8] https://medium.com/metamask/eip712-is-coming-what-to-expect-and-how-to-use-it-bb92fd1a7a26
[9] https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule
[10] https://solidity.readthedocs.io/en/develop/bugs.html

Rounding Errors

(Unsigned) integer divisions generally suffer from rounding errors. The same holds true for divisions inside the EVM. Therefore, the results of arithmetic operations can be imprecise. The effects of these errors can be reduced by ordering arithmetic operations in a numerically stable manner. However, even then minor errors (e.g. in the order of one token wei) can occur.

Disclaimer

Contact:

ChainSecurity AG
Krähbühlstrasse 58
8044 Zurich, Switzerland
contact@chainsecurity.com