

Security Audit

of MARKET PROTOCOL Smart Contracts

May 22, 2019






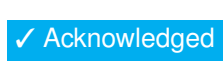

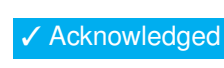

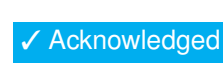




Produced for



by



Table Of Content

Foreword	1
Executive Summary	1
Audit Overview	2
1. Scope of the Audit	2
2. Depth of Audit	2
3. Terminology	2
4. Limitations	3
System Overview	4
1. Walkthrough	4
Best Practices in MARKET PROTOCOL's project	5
1. Hard Requirements	5
2. Best Practices	5
3. Smart Contract Test Suite	5
Security Issues	6
1. SafeERC20 outdated/flawed version  	6
2. withdrawFees not using safeTransfer  	6
3. Locked token  	7
Trust Issues	8
1. Owner could possibly drain all collateral tokens  	8
2. Usage of internal oracle  	8
3. Upgradeable MKT token  	8
Design Issues	9
1. arbitrateSettlement no price cap/floor checks  	9

2.	Missing validation basis points arguments	M	✓ Acknowledged	9
3.	Missing authorization on <code>deployMarketContractMPX</code>	M	✓ Acknowledged	9
4.	Missing validation of <code>QTY_MULTIPLIER</code> in <code>MarketContract</code>	M	✓ Fixed	9
5.	Empty strings in <code>contractNames</code> allowed	M	✓ Fixed	10
6.	Repetitive calls to the same function	M	✓ Addressed	10
7.	Public functions in library	M	✓ Fixed	10
8.	Over-complicated design	M	✓ Fixed	11
9.	Unit and decimal mismatches	L	✓ Acknowledged	11
10.	Missing validation address arguments <code>MarketContractFactoryMPX</code> constructor	L	✓ Fixed	12
11.	Superfluous return variable	L	✓ Fixed	12
12.	Missing validation <code>oracleURL</code> field in <code>deployMarketContractMPX</code>	L	✓ Acknowledged	12
Recommendations / Suggestions				13
Addendum and general considerations				14
1.	Dependence on block time information			14
2.	Outdated compiler version			14
3.	Forcing ETH into a smart contract			14
4.	Rounding Errors			14
Disclaimer				15

Foreword

We first and foremost thank MARKET PROTOCOL for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

Here, CHAINSECURITY provides the executive summary of the MARKET PROTOCOL audit. CHAINSECURITY was tasked by MARKET PROTOCOL to audit the MARKET PROTOCOL smart contracts and produced this report. As part of this summary, CHAINSECURITY briefly states the purpose of the project and then summarizes the findings of the report.

The MARKET PROTOCOL platform offers the ability to issue derivatives, whereby a certain collateral token is exchanged for a pair of long and short tokens. Price settlement is done by an oracle (or MARKET PROTOCOL if necessary), which/who will call a function once to settle the contract. Once a derivative has been settled, the long and short tokens can be used to claim back the corresponding value of collateral. The smart contracts will be used through the MARKET PROTOCOL's UI. Thereby, MARKET PROTOCOL has the power to decide which derivatives are listed on their web platform.

During the code audit, CHAINSECURITY uncovered 2 high and 1 low severity security issues, as well as 12 low/medium severity design issues and 3 low/medium severity trust issues. All discovered issues were quickly fixed, addressed or acknowledged and hence, no security concerns remain at this time.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. They have been reviewed based on SOLC compiler, version 0.5.2 and EVM version PETERSBURG. All of these source code files were received on May 1, 2019 as part of the git commit 27a1ba317a0230fff7583741d49f76215169c6c0. The latest update has been received on May 20, 2019 with the commit 485670d723ff0f686bda3f6878a8a0f0a99ca449.

In Scope	File	SHA-256 checksum
	./MarketCollateralPool.sol	5f21b645ec4d0203055e5942827d1fce47a701b60f0f437ec0b003ea60f57d7b
	./MarketContract.sol	0265c95ccc11a0bb4a601973e87648c0d95f520eba9769e500cb37e6ab44ddb7
	./MarketContractRegistry.sol	8f0043acaae653428dffac5d251ce02da548180c7be47c8172e19e6aa1201ff
	./MarketContractRegistryInterface.sol	7255dae90b36785ae484cbad4ff3e1055fd7bbef68d7748a2aaa16efc057d93e
	./libraries/MathLib.sol	22bfc180a3f817709ee6ab7a228eed32f059b376eee8419a315dd446dae8d741
	./mpx/MarketContractFactoryMPX.sol	5907b773f4209f65ecf55f0e73fc495a6ad1e998aff5babe40d21f0b3be6908f
	./mpx/MarketContractMPX.sol	71c89b2a16f3f06d4bb85746be9b036f3c9270b7c37fc61a4bfc60a7462b4381
	./tokens/CollateralToken.sol	e9d03b49f5270b3b620f207b2b9e746d1ff86f939120db4480febeb6ec52b0a5
	./tokens/InitialAllocationCollateralToken.sol	c77fb6e6c92f0c16e216db016be08dd540545fe530f74f0b2f18f0ef8990adaa
	./tokens/MarketToken.sol	f549ac38f70ab837f54fb9330510e68565ab9c3e1e8f05a1d5f380108e127f65
	./tokens/PositionToken.sol	5a314eafa4570fd99b2d7b9243fde10e3ab2a4b4dae94129181391781904c1f9
	./tokens/UpgradableToken.sol	866b0d8c341aa82ecb578c6d341ae51a1fca4e937fc72d824d1af2751e04e1e7
	./tokens/UpgradeableTarget.sol	833f3516edfb2fcbfe09aa4b5fc6729cd13c15f219422d9e83a53d1e28958c7c
	./tokens/UpgradeableTokenMock.sol	01d56f3c91a0eca46a7092e882c31a6f258651f94d6365051b040955fed58489

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 of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

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

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology¹).





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.










¹https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Severity is derived based on the likelihood and the impact calculated previously.





We categorize the findings into 4 distinct categories, depending on their severities:



-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

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

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of 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 labelled as either  **Fixed** or  **Addressed** are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover 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 lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

MARKET PROTOCOL is a platform to issue derivatives. To issue a new derivative, first, a corresponding smart contract needs to be setup. Therefore, an investor defines the terms for the derivative. These terms include (not exhaustive): what the underlying is (defined by an oracle price feed for the underlying), name of the contract, expiration date, cap, floor, quantity multiplier, fees.

If a contract with the desired terms exists, any account can exchange a collateral versus long and short tokens (amount depends on the terms and the desired quantity).

By keeping the long token and selling the short token an issuer has a long exposure and vice versa. The same amount of short and long tokens have no market exposure but can be used to redeem the corresponding collateral. If the price falls below the floor or raises above the cap, the derivative is due immediately and settled. The settlement price is either the cap or floor in this case. If the price raises above the cap, the short position is worthless and if it falls below the floor, the long position is worthless. A contract is also settled when the contract expires or when MARKET PROTOCOL forces an early settlement.

The next section gives a short technical overview over MARKET PROTOCOL's smart contract. Basically there are these contracts to interact with:

- `MarketCollateralPool` (used to manage the collateral)
- `MarketContractRegistry` (used to manage the factory contract and whitelisted contracts)
- `MarketContractFactoryMPX` (used to set up new Market contracts)
- `MarketContractMPX` (used to define a derivative and issue the long and short tokens) inherits from `MarketContract`

All contracts are ownable. Additionally, the audited contracts included token contracts. Most important the `PositionToken` which are the long and short tokens. Each market contract has their own pair of position tokens.

Walkthrough

Assuming the collateral tokens and the MARKET PROTOCOL TOKENS already exist. MARKET PROTOCOL initializes the system by deploying the three main contracts. (1) The `MarketContractRegistry` contract, (2) the `MarketCollateralPool` contract and (3) the `MarketContractFactoryMPX`. The deployment also includes the deployment and linking of the corresponding libraries. To finalize the setup MARKET PROTOCOL needs to call some initializer and setter functions.

MARKET PROTOCOL or any user can now use the system and set up new derivatives by calling `MarketContractFactoryMPX.deployMarketContractMPX()` and providing all the terms as arguments. This will deploy a market contract with the corresponding long and short tokens. By calling `MarketCollateralPool.mintPositionTokens()` a user can deposit a collateral and get the corresponding long/short position tokens. These tokens can be traded at will or redeemed before the contract is settled by calling `MarketCollateralPool.redeemPositionTokens()`. The contract is also automatically settled by the oracle callback if the price crosses the cap/floor or when the contract is expired. When settled, a user can claim the collateral tokens in exchange for their position tokens by calling `MarketCollateralPool.settleAndClose()`.

The only role with power is MARKET PROTOCOL with the owner account. The owner can withdraw the fees and call the setter functions. The owner can also call `MarketContractMPX.arbitrateSettlement()` to immediately settle a contract and provide an arbitrary settlement price.

Best Practices in MARKET 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 fulfillment of general guidelines.

Hard Requirements

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

- ✓ **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.
- ✓ 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.
- ✓ There are migration/deployment scripts executable by CHAINSECURITY and their use is documented.
- ✓ The code is provided as a Git repository to allow the review 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.
- ✓ Code duplication is minimal, or justified and documented.
- ✓ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ✓ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- ✓ There is no dead code.
- ✓ The code is well documented.
- ✓ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ✓ Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- ✓ Function are grouped together according to either the Solidity guidelines², or to their functionality.

Smart Contract Test Suite

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

The MARKET PROTOCOL project contains 84 truffle tests and has an overall test coverage above 90% at the time of writing. CHAINSECURITY is very pleased with the amount of tests, coverage level and code style of the tests.

Next to testing for success cases, MARKET PROTOCOL also tests for failure cases, and in case of failure cases the error message is checked.

²<https://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions>

Security Issues

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

SafeERC20 **outdated/flawed version**  

In practice there are two types of ERC20 implementations.

- Some older ERC20 tokens do not provide any return value when functions such as `transferFrom()` are called. Among, these tokens, there are some popular ones such as OmiseGo³.
- Token contracts with a correct implementation of the standard, return a **bool** value to let the caller know about the status of the transfer.

Some tokens not returning a boolean has been a known issue and caused problems⁴ with Decentralized Exchanges (DEXs).

In OpenZeppelin release 2.2.0⁵ the SafeERC20 contract was refactored to take into account tokens that return nothing on success. MARKET PROTOCOL is currently using OpenZeppelin 2.0.1 which will revert if a token function returns no **bool** on success. CHAINSECURITY notes that OpenZeppelin 2.2.0 does not work with solidity version 0.4.x. Therefore, MARKET PROTOCOL might have to upgrade to solidity version 0.5.x.

MARKET PROTOCOL is advised to make use of the updated SafeERC20 contract as provided by OpenZeppelin version 2.2.0 and above.

Likelihood: Medium

Impact: High

Fixed: MARKET PROTOCOL upgraded openzeppelin—solidity to version 2.2.0.

withdrawFees **not using** safeTransfer  

MARKET PROTOCOL makes use of the OpenZeppelin SafeERC20 library to safely call functions of ERC20 tokens that return false on error, instead of throwing. However, there is one place where a regular transfer call is executed instead of the safe variant (`safeTransfer`), see line 234 in `MarketCollateralPool`.

```
228 function withdrawFees(address feeTokenAddress, address feeRecipient) public
    onlyOwner {
229     uint feesAvailableForWithdrawal = feesCollectedByTokenAddress[
        feeTokenAddress];
230     require(feesAvailableForWithdrawal != 0, "No fees available for withdrawal")
231     require(feeRecipient != address(0), "Cannot send fees to null address");
232     feesCollectedByTokenAddress[feeTokenAddress] = 0;
233     // EXTERNAL CALL
234     ERC20(feeTokenAddress).transfer(feeRecipient, feesAvailableForWithdrawal);
235 }
```

MarketCollateralPool.sol

If the ERC20 contract from where to extract the fee does not throw, but returns a boolean to indicate failure/success, then in case the transfer was unsuccessful, the function would return `false`. Since, the return value of the transfer call is not checked inside `withdrawFees`, the function will silently continue as if the transfer was successful. The amount of tokens to send to the fee recipient is reset to zero before doing the transfer call. Thus, it appears as the fees were paid, but they are not and are now locked in the token contract.

³<https://etherscan.io/address/0xd26114cd6EE289AccF82350c8d8487fedB8A0C07#code>

⁴<https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca>

⁵<https://github.com/OpenZeppelin/openzeppelin-solidity/releases/tag/v2.2.0>

MARKET PROTOCOL is advised to check the `transfer` call return value, and correctly handling a failed call.

Likelihood: Medium

Impact: High

Fixed: MARKET PROTOCOL now uses `SafeERC20.safeTransfer` instead of `ERC20.transfer`.

Locked token



✓ Acknowledged

If tokens, especially MKTs, are accidentally or intentionally sent to the market or collateral pool contract, the tokens will be locked, with no way to recover them. Incidents within the past (but with ERC20 tokens) showed this is a real issue as there always are users sending tokens to the token contract due to unknown reasons. See here for some examples: ⁶.

Note: ETH and tokens can be forced into any contract and be locked there if no "recover" function exists. If contracts are not supposed to handle/process Ether or token transfers, CHAINSECURITY does not further report this kind of locked tokens. CHAINSECURITY reports locked token or locked ETH for contracts which are supposed to or where the user could think these contracts might handle/process tokens or ETH in some way.

Likelihood: Low

Impact: Low

Acknowledged: MARKET PROTOCOL acknowledged the issue. To avoid trust issues they decided to leave it as it is.

⁶<https://coincentral.com/erc223-proposed-erc20-upgrade/>

Trust Issues

This section mentions functionality which is not fixed inside the smart contract and hence requires additional trust into MARKET PROTOCOL, including in MARKET PROTOCOL's ability to deal with such powers appropriately.

Owner could possibly drain all collateral tokens

The owner has the ability to add any address to the MarketContractRegistry whitelist. The owner can therefore, whitelist a custom contract which would allow draining of all tokens of a specific collateral token. Users need to trust the owner, to not exercise this functionality to perform the above actions.

MARKET PROTOCOL is advised to re-evaluate the owner-allowed actions in MarketContractRegistry.

Acknowledged: MARKET PROTOCOL acknowledged the issue.

Usage of internal oracle

The MARKET PROTOCOL project makes use of oracles to provide necessary pricing data coming from external sources. According to the whitepaper, these oracles will be provided by a third party service such as ChainLink or BlockOne IQ, or an oracle will be developed specifically for a contract. The provided repository, however, does not contain any oracle related contracts.

After inquiry, MARKET PROTOCOL made clear, it will be using only an "internal" oracle for the initial launch. CHAINSECURITY has the opinion this should be explicitly mentioned in the whitepaper. Users might think some specialized third party will provide the oracle. This is not the case. The oracle is not independent, nor provided by a specialized provider. Furthermore, in the two cases a user needs to trust different entities. Hence, CHAINSECURITY's opinion is, this is valuable information for potential users and needs to be mentioned.

Acknowledged: MARKET PROTOCOL has pending Whitepaper updates.

Upgradeable MKT token

The MKT token implements the functionality for token upgrade which CHAINSECURITY did not find mentioned in the whitepaper. According to the implemented upgrade scheme, when the MKT token is upgraded, the new version will exist at a different address and users are expected to call upgrade on the original token contract, which will move their tokens to the new token contract.

CHAINSECURITY does not see a problem with this upgrade scheme. CHAINSECURITY recommends that this functionality is described in the documentation, including the whitepaper, to inform token holders about it.

Acknowledged: MARKET PROTOCOL has pending Whitepaper updates.

Design Issues

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

`arbitrateSettlement` no price cap/floor checks

Besides the oracle calling the `oracleCallback` function, there is also an `arbitrateSettlement` function to force immediate settlement by the owner, regardless of the expiration time of the `MarketContract`. Although the oracle callback will check if the reported price is above/below the cap/floor, there are no such checks when `arbitrateSettlement` is executed. This could lead to the settlement price being below/above the configured cap/floor.

Because only the owner can call this function the users need to trust the owner to stay within the floor/cap. However, this issue could be completely prevented by also applying the cap/floor checks when executing `arbitrateSettlement`.

CHAINSECURITY advises MARKET PROTOCOL to add the cap/floor checks when `arbitrateSettlement` is executed.

Fixed: MARKET PROTOCOL added cap/floor checks.

Missing validation basis points arguments

The constructor of `MarketContract` takes in a number of arguments. Two of these represent the percentage to use as fee when users mint new position tokens, `feeInBasisPoints` and `mktFeeInBasisPoints`. The maximum percentage is 100, which is represented as 10000, meaning 100.00%. Therefore, MARKET PROTOCOL is advised to add checks to make sure these two values are at most 10000.

Fixed: MARKET PROTOCOL acknowledged the issue, but did no code changes.

Missing authorization on `deployMarketContractMPX`

To add a new Market Contract, anybody can call the `MarketContractFactoryMPX.deployMarketContractMPX` function. This function will deploy a new `MarketContractMPX` contract and add its address to the `MarketContractRegistry` whitelist.

MARKET PROTOCOL has stated, they will only show market contracts in the application which have been explicitly created by MARKET PROTOCOL. Therefore, CHAINSECURITY sees no reason for anybody being able to call the `deployMarketContractMPX` function. Calling this function allows anybody to add new market contracts to the whitelist in `MarketContractRegistry`.

MARKET PROTOCOL is advised, to reevaluate who is allowed to call `deployMarketContractMPX`.

Acknowledged: MARKET PROTOCOL acknowledged, future use cases may dictate allowing this.

Missing validation of `QTY_MULTIPLIER` in `MarketContract`

The `QTY_MULTIPLIER` variable in the `MarketContract` constructor is not validated. If this value is zero, the amount of collateral that is exchanged for one short plus one long token will be zero.

CHAINSECURITY believes that this behavior is unintended. MARKET PROTOCOL is advised, to add a check which prevents this variable from being zero.

Fixed: MARKET PROTOCOL added a check.

Empty strings in contractNames allowed

The first argument to the `MarketContractFactoryMPX.deployMarketContractMPX` function is **string** `contractNames`. This function can be called by anybody. Calling this function will deploy a new `MarketContract`. According to the function comment this variable contains: `contractName, longTokenSymbol, shortTokenSymbol`

```
104 StringLib.slice memory pathSlice = contractNames.toSlice();
105 StringLib.slice memory delim = ",".toSlice();
106 require(pathSlice.count(delim) == 2, "ContractNames must contain 3 names");
107 CONTRACT_NAME = pathSlice.split(delim).toString();
108
109 PositionToken longPosToken = new PositionToken("MARKET Protocol Long Position
      Token", pathSlice.split(delim).toString(), 0);
110 PositionToken shortPosToken = new PositionToken("MARKET Protocol Short
      Position Token", pathSlice.split(delim).toString(), 1);
```

MarketContract.sol

The `contractNames` variable is expected to contain three comma-separated strings, for example `ETHUSD, LONG, SHORT`. However, due to the current validation it is allowed to pass in empty values for all three strings by sending `,,` as the value for `contractNames`. The result is an empty string as the value of `CONTRACT_NAME`, the short token name and the long token name.

Also, CHAINSECURITY is wondering why MARKET PROTOCOL chose to pack these three names inside one comma-separated string instead of using three separate string variables. If using separate variables, the costly string manipulation would be unnecessary and the validation would be much simpler. This is also referenced in another issue about some inefficient and over-complicated implementations regarding the libraries.

CHAINSECURITY advises MARKET PROTOCOL, to reevaluate the implementation of `contractNames`.

Fixed: MARKET PROTOCOL refactored the implementation and no longer makes use of the string splitting code.

Repetitive calls to the same function

Inside `MarketCollateralPool.mintPositionTokens()` there are several functions of the `MarketContract` which are called multiple times, each time with the same arguments (=none) and therefore, returning the same result.

It is considered best practice to call the function once, save the result in a variable, and use this variable whenever the result is needed in the rest of the function. This will lower the gas cost of executing this function. As it costs more to call a function than retrieve a value stored a local variable.

A list of `MarketContract` functions which are called multiple times within `mintPositionTokens()`:

- `marketContract.COLLATERAL_TOKEN_ADDRESS()`
- `marketContract.MKT_TOKEN_FEE_PER_UNIT()`
- `marketContract.COLLATERAL_TOKEN_FEE_PER_UNIT()`

MARKET PROTOCOL is advised to call each function once and use a local variable to store the result.

Addressed: MARKET PROTOCOL explained that there is no viable solution. The stack will get too deep if they use more variables.

Public functions in library

MARKET PROTOCOL uses public functions in their `MathLib` library contract. However, these functions could also be defined as **internal**. In that case all of the library's functions would be **internal**. This which would make the solidity compiler inline the library in the contracts⁷.

⁷<https://medium.com/coinmonks/all-you-should-know-about-libraries-in-solidity-dd8bc953eae7>

This would mean the library would not be separately deployed (and thus no need for truffle "linking"). Also, the gas cost of executing any library provided function would be lower, e.g. `settleAndClose` would be 2000+ gas cheaper.

MARKET PROTOCOL is advised to reevaluate the use of `public` vs `internal` functions inside `MathLib`.

Fixed: MARKET PROTOCOL made the functions `internal`.

Over-complicated design

This issue summarizes two issues regarding functionality which could be designed simpler, less error prone, and more efficient. The two functionalities are: signed integers and string splitting. For this, MARKET PROTOCOL uses the libraries `MathLib` and `StringLib` respectively.

Signed Integers

Signed integers are only used in the `MarketCollateralPool.settleAndClose` function (and `MathLib.calculateNeededCollateral` which is called from `settleAndClose`). Inside the `settleAndClose` function the `qtyRedeem` argument is of type `int`. A positive value represents long tokens to settle, and a negative value represents short tokens to settle. Therefore, an account can not settle both long and short tokens in one transaction, but needs two transactions. Additionally, this adds extra code to do safe signed integer arithmetic.

MARKET PROTOCOL could instead of one `int` argument, use two `uint` function arguments, the long and short token amount respectively. If MARKET PROTOCOL were to do this, there would be no need for any signed integer arithmetic in this codebase and the corresponding code could be removed. This would also make it possible for an account to settle both his long and short tokens in one transaction.

String Manipulation

String splitting is used in the constructor of `MarketContract`. A `string` variable is splitted in three using a comma as separator. MARKET PROTOCOL could also use three separate variables. In that case there is no need for `StringLib`. Which could be removed from the codebase.

If MARKET PROTOCOL would follow up on these two recommendations, the complexity and attack surface of the smart contracts would decrease. Furthermore, the gas cost of deploying the contracts, as well as the gas cost of executing functions in the contracts decrease.

Fixed: MARKET PROTOCOL removed the signed integer code and no longer splits a string.

Unit and decimal mismatches

A collateral in a market contract can be any token contract. Given this token is a proper implemented ERC20 token, it has a specified `decimals` value. This often is 18 but could also be any other value.

The price provided by the price feed has also decimals (originally). This price needs to be converted to a `uint` when passing it through the oracle callback into the contract. The more precise a price needs to be, the more decimals it needs. Hence, it will result in a bigger `uint`.

The collateral which needs to be put down is the difference between the cap and the floor. Hence, the bigger the difference the more collateral needs to be provided. As mentioned before, an accurate price needs big integer values. Cap and floor, therefore, will also be big integers. Hence, the difference between cap and floor is a big integer. This could end up problematic if the collateral token has not enough decimals. The values are linked together. The following example will illustrate the problem.

Underlying price feed: USD/EURO - price 0.885542

The market contract specs are: cap = 1, floor = 0.75, Multiplier = 1, Collateral token: Gimini (<https://etherscan.io/token/0x056fd409e1d7a124bd7017459dfea2f387b6d5cd#readContract>) (decimals 2)

We cannot use floating point numbers in the contract hence, we need to convert the price, the cap and the floor. This is: price = 885542, cap = 1000000 and floor=750000. One unit of position tokens (0.00001)⁸ is now cap-floor=1000000-750000. Thus, 250000 units of collateral token are needed. This is no problem if a collateral token has enough decimals. In most cases this will be a super small amount. But in our example, the token has just 2 decimals. Hence, one unit is 0.01 token. We therefore need, 2500 Token. Gimini is a stable coin with a peg around 1 USD. To collateralize this position 2500 USD are needed. This can get way worse the less decimals the collateral token has or the more accurate the price feed is. It is easy to see, that

⁸There is a separate note on the meaning of a unit in the note section.

these values are linked and can quickly end up problematic. The multiplier could be used to compensate in the opposite direction but does not help in the example above.

MARKET PROTOCOL should be aware of this issue and prevent this from happening. Either by code or by (1) communicate this problem to future customers and (2) carefully chose the collateral⁹

Acknowledged: MARKET PROTOCOL acknowledged the issue. No changes were made.

Missing validation address arguments `MarketContractFactoryMPX` constructor

The constructor of `MarketContractFactoryMPX` takes in three address arguments. There is no validation to prevent these addresses from being `address(0)`. The first and third arguments are stored in a variable which can later be updated. However, the second one can not be updated afterwards. If this variable is incorrect, none of the mint and redeem functions on the market contract will be callable. Effectively making this contract useless.

The solution is to simply redeploy. Still, CHAINSECURITY thinks MARKET PROTOCOL could guard for this by preventing any of the arguments from being address zero.

Fixed: MARKET PROTOCOL now checks for zero address.

Superfluous return variable

The function `removeContractFromWhiteList()` in `MarketContractRegistry` returns a bool. This bool is never explicitly set and therefore, will always return false. MARKET PROTOCOL could consider not returning anything from this function.

Fixed: MARKET PROTOCOL removed the return variable.

Missing validation `oracleURL` field in `deployMarketContractMPX`

The `oracleURL` argument in the `MarketContractFactoryMPX.deployMarketContractMPX` function is not validated in any way and can, therefore, also be an empty string. As this field is necessary for the oracle to work, CHAINSECURITY thinks there should be at least validation to prevent this argument from being an empty string.

Acknowledged: MARKET PROTOCOL acknowledged, they will check this off-chain.

⁹CHAINSECURITY also put a note into the addendum, regarding the choice of collateral and possible implications.

Recommendations / Suggestions

- ☐ The function `transferOwnership()` does transfer 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 only argument in the `UpdatedLastPrice` event defined inside `MarketContract` is the new price. It is set by the oracle. MARKET PROTOCOL could also consider adding the previous price.
- ☒ The `calculateNeededCollateral` function inside `MathLib` is used to calculate the value to be stored in the variable `collateralToReturn`. Therefore, CHAINSECURITY suggests MARKET PROTOCOL re-names the function to `calculateCollateralToReturn`.
- ☐ MARKET PROTOCOL uses `mixedCase` and `UPPER_CASE_WITH_UNDERSCORES` as variable names in their contracts. The Solidity style guidelines¹⁰ recommend that the variable names should be mixed-Case and start with a lower case letter. The uppercase variant should only be used for variables marked `constant`¹¹.
- ☐ The comment for the `oracleStatistic` argument reads: `statistic type (lastPrice, vwap, etc)`. If the values can only be one of a known set of values, MARKET PROTOCOL could consider using an enum for this variable.
- ☒ Inside `PositionToken` a `uint8` variable named `MARKET_SIDE` is defined. The value of this variable is passed in from the `MarketContract` constructor and will always be either 0 or 1, representing Long and Short respectively. In another contract this variable is defined as an enum, see `MarketCollateralPool`. CHAINSECURITY suggests client normalizes the type of the market side variable throughout the contracts, preferably an enum.
- ☒ MARKET PROTOCOL could consider to add error messages to `require()` statements, to indicate why a call failed. MARKET PROTOCOL's `safeMath` library is inspired by `OpenZeppelin`, but does not provide error messages like OZ. There are multiple `require` statements throughout the contracts where no error message has been defined.
- ☒ It is possible to enable and configure the `solc` optimizer inside the `truffle.js` configuration file. Using the optimizer results in more gas efficient deployment or more gas efficient calls to the contracts. MARKET PROTOCOL could consider adding this inside `truffle.js`.
- ☐ The array `addressWhiteList` is not used in the code base. CHAINSECURITY assumes it is needed for the dApp or similar applications. If not needed, MARKET PROTOCOL could consider removing the array.

¹⁰<https://solidity.readthedocs.io/en/v0.4.25/style-guide.html#local-and-state-variable-names>

¹¹<https://solidity.readthedocs.io/en/v0.4.25/style-guide.html#constants>

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.

CHAINSECURITY mentions general issues in this section which are relevant for MARKET PROTOCOL's code, but do not require a fix. Additionally, CHAINSECURITY mentions information in this section, to clarify or support the information in the security report. This section, therefore, serves as a reminder to create awareness for MARKET PROTOCOL and potential users.

Dependence on block time information

MARKET PROTOCOL uses `now` inside the `MarketContract` 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 issue¹².

Outdated compiler version

CHAINSECURITY could not find obvious issues with the compiler version MARKET PROTOCOL is using. While the latest compiler release is version 0.5.8, MARKET PROTOCOL uses SOLC compiler, version 0.5.2. If MARKET 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 bugs¹³.

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 contract. 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.

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.

¹²<https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule>

¹³<https://solidity.readthedocs.io/en/develop/bugs.html>

Disclaimer

UPON REQUEST BY MARKET PROTOCOL, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..