BLOCKSEC

Security Audit Report for Mage Finance Smart Contracts

Date: April 7, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Mage Finance
Target	Mage Finance Smart Contracts

Version History

Version	Date	Description
1.0	March 18, 2024	First Release
1.1	April 7, 2024	Update for Oracle

Signature

Digitally signed by: BlockSec (https://blocksec.com)

Email: audit@blocksec.com

Date: 2024-04-08 02:15:27+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is the contracs of Mage Finance [1]. Mage Finance is a lending protocol forked from the famous Compound Protocol [2] on the Merlin blockchain, with custom oracles that depend on Chainlink and Pyth Oracles. Please note that only the contracts inside the `contracts` folder in the repository are within the scope of this audit. Test-related files, specifically the files in the `contracts/Mock` directory, are not included in the scope of the audit.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Mage Finance Smart Contracts	`Version 1`	`7c6b7c4281b8c6fa1769d8afe3682d5d0573d3fc`
	`Version 2`	`5e5772169ec639c932cff9e96fc5bb8f5d54b21d`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] `https://github.com/MageFinance/mage-contracts`

[2] `https://github.com/compound-finance/compound-protocol`

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

∗ Reentrancy
∗ DoS
∗ Access control
∗ Data handling and data flow
∗ Exception handling
∗ Untrusted external call and control flow
∗ Initialization consistency
∗ Events operation
∗ Error-prone randomness
∗ Improper use of the proxy system

1.3.2 DeFi Security

∗ Semantic consistency
∗ Functionality consistency
∗ Permission management
∗ Business logic
∗ Token operation
∗ Emergency mechanism
∗ Oracle security
∗ Whitelist and blacklist
∗ Economic impact
∗ Batch transfer

1.3.3 NFT Security

∗ Duplicated item
∗ Verification of the token receiver
∗ Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **one** potential security issue. Besides, we also have **four** notes.

- High Risk: 1
- Note: 4

ID	Severity	Description	Category	Status
1	High	Flawed price oracle implementation	Software Security	Fixed
2	-	Potential centralization risk	Note	-
3	-	Potential `exchangeRate` manipulation due to empty markets	Note	-
4	-	Limitation on the `Multicall` contract	Note	-
5	-	Potential failure on native token transfer	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Flawed price oracle implementation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Like Compound, Mage Finance relies on external oracles to provide price information used in the calculation of account liquidity. It utilizes two oracles, Chainlink and Pyth, to provide external price data. As external oracles, both Chainlink and Pyth provide information about the prices returned, such as the timestamp of the latest price update and valid price ranges. It is crucial for any project that relies on the prices provided by these oracles to check the validity of the prices to ensure that only timely and accurate prices are used. However, the smart contract implementations in Mage Finance that query prices from these two oracles are flawed, as presented below:

- The following code snippet shows the implementation of the `ChainlinkOracle` contract. It is clear that there are no further checks on the validity of the price provided by Chainlink.

```
function getOraclePrice(address token) internal view override returns (uint price) {
    (, int256 _price) = IAggregator(priceFeeds[token]).latestRoundData();
    price = uint256(_price);

    uint decimalDelta = uint(18) - USD_DECIMAL;
    if (decimalDelta > 0) {
        price = price * (10 ** decimalDelta);
    }
}
```

Listing 2.1: Oracle/ChainlinkOracle.sol

- The following code snippet is the implementation of the PythOracle contract. There are no further checks on the price returned by Pyth either.

```solidity
function getOraclePrice(address token) internal view override returns (uint price) {
    Price memory _price = pyth.getPriceUnsafe(priceFeeds[token]);
    price = uint(uint64(_price.price));

    uint decimalDelta = uint(18) - USD_DECIMAL;
    if (decimalDelta > 0) {
        price = price * (10 ** decimalDelta);
    }
}
```

Listing 2.2: Oracle/PythOracle.sol

Apart from the lack of validity checks, there is another potential inconsistency problem in the oracle implementations presented above. Currently, both oracles use a constant (i.e., USD_DECIMAL = 8) to scale up the prices. If all the prices used in Mage Finance are based on the USD value, then the calculation can be simplified to a single constant. Otherwise, the precision should be read from the oracle contracts for consistency.

Impact The protocol may use outdated and incorrect prices due to insufficient checks.

Suggestion Check the timeliness and correctness of the prices in the code implementation.

2.2 Note

2.2.1 Potential centralization risk

Description Compared with the original Compound Protocol, Mage Finance introduces several aspects that increase the risk of centralization, which are listed as follows:

- First, there is a mechanism called the "direct prices" within the BaseOracle contract, where the contract admin can directly set a price for a given token. These "direct prices" take priority over prices read from oracles, allowing the contract admin to overwrite the price for any token.

```solidity
function getPriceByToken(address token) public view returns (uint price) {
    uint directPrice = directPrices[token];
    if (directPrice > 0) {
        price = directPrice;
    } else {
        price = getOraclePrice(token);
    }
```

Listing 2.3: Oracle/BaseOracle.sol

- Second, the getUnderlyingPrice function in the BaseOracle contract uses symbol from the CToken contract to determine whether the underlying asset of CToken is the native token. However, there is a possibility that a malicious CToken with a native symbol could be used to mimic the native token, causing the malicious CToken to be overpriced within Mage Finance.

```
function getUnderlyingPrice(Token cToken) external view returns (uint) {
    string memory symbol = cToken.symbol();
    if (compareStrings(symbol, nativeSymbol)) {
        return getOraclePrice(NATIVE_ADDRESS);
    } else {
        return getPrice(cToken);
    }
}
```

Listing 2.4: Oracle/BaseOracle.sol

2.2.2 Potential `exchangeRate` manipulation due to empty markets

Description As a lending protocol forked from the Compound Protocol, Mage Finance inherits problems that may be present in the original implementation. Like Compound, it is susceptible to potential `exchangeRate` manipulation in empty markets[1]. However, this issue can be addressed by adding initial liquidity to a newly created market.

2.2.3 Limitation on the `Multicall` contract

Description In the smart contracts of Mage Finance, there is a utility contract called `Multicall` that can initiate multiple calls in a single transaction. It should be ensured that this utility contract is not allocated any assets and is granted no special roles to ensure the safety of the entire protocol.

2.2.4 Potential failure on native token transfer

Description Because Mage Finance is forked from the Compound Protocol, it inherits a potential caveat in the `CEther` implementation of the Compound Protocol, as `CEther` uses plain `transfer` for transferring native tokens back to users when they are redeeming. However, by default the Solidity `transfer` will only allocate 2,300 gas which is sufficient for transferring native tokens to EOAs. When transferring native tokens to contracts, however, this amount of gas may be insufficient and could lead to potential failures of the token transfers.

```
function doTransferOut(address payable to, uint amount) internal virtual override {
    /* Send the Ether, with minimal gas and revert on failure */
    to.transfer(amount);
}
```

Listing 2.5: CEther.sol

[1] https://www-dev.blocksec.com/blog/6-hundred-finance-incident-catalyzing-the-wave-of-precision-related-exploits-in-vulnerable-forked-protocols

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS