Security Audit Report for Mobius Contracts

Date: Nov 16, 2021

Version: 1.1

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Mobius Finance
Target	Mobius Contracts

Version History

Version	Date	Description
1.0	Nov 14, 2021	First Release
1.1	Nov 16, 2021	Second Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

The target contract is Mobius Contracts. The detailed description is in the following link: Mobius Finance.

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

The files that are audited in this report include the following ones.

Repo Name	Github URL
mobius-contracts	

The commit hash before the audit is `8e25ab8e8d352ef759b70ed97e29daa96ee2ee08`. The commit hash that fixes the issues found in this audit is `28046a854201ebc128fa861d268705f147d52622`.

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 do 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.

1.3 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
- Access control
- 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 [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

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

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

Chapter 2 Findings

In total, we find seven potential issues, four recommendations and one concern in Mobius Contracts, as follows:

- Medium Risk: 1
- Low Risk: 2
- Undetermined Risk: 4
- Recommendation: 4
- Other Concern: 1

ID	Severity	Description	Category
1	Medium	*Potential Dos Vulnerability*	Software Security
2	Low	*Lack of Status Check*	Software Security
3	Low	*Potential Out-of-gas Problem*	Software Security
4	Undetermined	*Functionality Inconsistency in* `Issuer`	DeFi Security
5	Undetermined	*Unlimited Token Mint*	DeFi Security
6	Undetermined	*Unfair Global Debt Mechanism*	DeFi Security
7	Undetermined	*Unclear Trading Fee Distribution Mechanism*	DeFi Security
8	-	*Code Typos*	Recommendation
9	-	*Formula Inconsistency*	Recommendation
10	-	*Redundant Function*	Recommendation
11	-	*Do Not Use Elastic Tokens*	Recommendation
12	-	*Potential Front-running Due to the Price Oracle*	Other Concern

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential DoS Vulnerability

Status Confirmed and fixed.

Description There exists a potential DoS Vulnerability in the liquidation process. Part of the execution sequence of liquidation is: `Mobius.liquidate()` -> `Issuer.burnDebt()` -> `Issuer._burnDebtForUser()` -> `Issuer._canBurn()`. Specifically, line 157 in `Issuer._burnDebtForUser()` checks `lastTime` (the last time `Mobius._mint()` is invoked). As a result, if `Setting.getMinStakeTime()` is not zero, a user can regularly invoke `Mobius._mint()` to update `lastTime` so that any liquidation operation will fail because of the `_canBurn()` check.

```
149    function _burnDebtForUser(
150        bytes32 stake,
151        address account,
152        bytes32 debtType,
153        uint256 amount
154        ) internal returns(uint256,uint256,uint256) {
155            (uint256 accountDebtWithDynamic,uint256 accountDebt, uint256 originalDebt, uint256
                   lastTime) = _getDebt(stake, account, debtType);
156
```

```
        require(_canBurn(lastTime), 'Issuer: Minimum stake time not reached');
    ...
```

Listing 2.1: _burnDebtForUser:Issuer.sol

```
function _canBurn(uint256 time) private view returns (bool) {
    return block.timestamp >= time + (Setting().getMinStakeTime());
}
```

Listing 2.2: _canBurn:Issuer.sol

Impact In brief, a malicious user could constantly refresh his `lastTime` to block **ANY** liquidation operation.

Suggestion Bypass time check while liquidation.

2.1.2 Lack of Status Check

Status Confirmed and fixed.

Description The second part of return values (i.e., "status") of `AssetPrice.getPriceAndStatus()` is used to check whether `Synth` is online or not. However, the status is only checked in `Trader`, but not in `Issuer`.

```
function getPriceAndStatus(bytes32 asset) public override view returns (uint256, uint256) {
    if (asset == USD || asset == 'USDT') return (1 ether, 0);
    (, uint256 price, uint256 updateTime) = getPriceFromOracle(asset);
    require(price > 0, contractName.concat(': Price is zero For ', asset));

    uint256 lastTime = block.timestamp - maxDelayTime;
    if (updateTime < lastTime) return (price, 1);
    return (price, 0);
}
```

Listing 2.3: getPriceAndStatus:AssetPrice.sol

Impact May leard to inconsistent problem.

Suggestion Check status in `Issuer` as well.

2.1.3 Potential Out-of-Gas Problem

Status Acknowledged. However, as stated by the developers: "there is no better way to calculate the global debt".

Description The implementations of `Issuer.getDynamicTotalDebt()`, `AssetPrice.getPrices()` and `MobiusOracle.setPrices()` iterate all synth assets and calculate the global debt of the platform, which may result in out-of-gas exception while execution.

```
function getDynamicTotalDebt() public override view returns (uint256 platTotalDebt, uint256
     usersTotalDebt, uint256 usersTotalDebtOriginal) {
    bytes32[] memory synths = assets(SYNTH);
    uint256[] memory prices = AssetPrice().getPrices(synths);

    for (uint256 i = 0; i < synths.length; i++) {
        (uint256 debt,uint256 debtOriginal ,) = Storage().getTotal(synths[i]);

```

```
        usersTotalDebtOriginal = usersTotalDebtOriginal + debtOriginal;
        usersTotalDebt = usersTotalDebt + debt.decimalMultiply(prices[i]);

        address synth = requireAsset(SYNTH, synths[i]);
        platTotalDebt = platTotalDebt + (IERC20(synth).totalSupply().decimalMultiply(prices[i])
            );
    }
}
```

Listing 2.4: getDynamicTotalDebt:Issuer.sol

```
function getPrices(bytes32[] calldata assets) external override view returns (uint256[] memory
    ) {
    (uint256[] memory prices, ) = getPricesAndStatus(assets);
    return prices;
}
```

Listing 2.5: getPrices:AssetPrice.sol

```
function setPrices(bytes32[] calldata assets, uint256[] calldata prices) external override
    allManager {
    require(assets.length == prices.length, 'MobiusOracle: asset and price length mismatch');
    for (uint256 i = 0; i < assets.length; i++) {
        setPrice(assets[i], prices[i]);
    }
}
```

Listing 2.6: setPrices:MobiusOracle.sol

Impact The transaction may be reverted due to out-of-gas.

Suggestion N/A

2.2 DeFi Security

2.2.1 Functionality Inconsistency in `Issuer`

Status Confirmed and fixed.

Description Both `issueDebt()` and `issueDebtWithPreviousStake()` of `Issuer` issue debts, the former provides the rewards (trade mining) while the latter does not, which is unreasonable.

```
function issueDebt(
    bytes32 stake,
    address account,
    bytes32 debtType,
    uint256 amountInUSD,
    uint256 amountInSynth
) external override onlyAddress(CONTRACT_MOBIUS) {

    _issueDebt(stake, account, debtType, amountInUSD, amountInSynth);

    if (debtType != USD) {
        (uint256 long, uint256 short) = DynamicTradingFee().getPositionInfo(debtType);
```

```
        if (long > short) {
            RewardTrading().tradeMining(account, requireAsset('Synth',USD), requireAsset('Synth
                ',debtType), amountInUSD * 5);
        } else {
            RewardTrading().tradeMining(account, requireAsset('Synth',USD), requireAsset('Synth
                ',debtType), amountInUSD);
        }
    }
}
```

Listing 2.7: issueDebt:Issuer.sol

```
function issueDebtWithPreviousStake(bytes32 stake, address account, bytes32 debtType, uint256
    amountInSynth) external override onlyAddress(CONTRACT_MOBIUS) {
    require(amountInSynth <= getIssuable(stake, account, debtType),"issueDebtWithPreviousStake:
        can not issue that much");
    uint256 debtPrice = AssetPrice().getPrice(debtType);
    _issueDebt(stake, account, debtType, amountInSynth.decimalMultiply(debtPrice),
        amountInSynth);
}
```

Listing 2.8: issueDebtWithPreviousStake:Issuer.sol

Impact The functionalities of `issueDebt()` and `issueDebtWithPreviousStake()` are inconsistent.

Suggestion Add rewarding mechanism for `issueDebtWithPreviousStake()`.

2.2.2 Unlimited Token Mint

Status Not an issue. This is guaranteed by the developers: "`deposit()` is for Polygon bridge and can only be called by Polygon childchain admin, while airdrop() is the first thing we did after TGE (actually this function is bounded by totalSupply() since our token has launched)".

Description `deposit()` and `airdrop()` are not bound by `MAX_SUPPLY` in `MobiusToken` (i.e., `deposit()` and `airdrop()` could mint arbitrary amounts of tokens).

```
function deposit(address user, bytes calldata depositData) external {
    require(msg.sender == DEPOSITOR_ROLE, "caller is not DEPOSITOR_ROLE");
    uint256 amount = abi.decode(depositData, (uint256));
    _mint(user, amount);
}
```

Listing 2.9: deposit:MobiusToken.sol

```
function airdrop(address to,uint256 amount) external onlyOwner returns (bool) {
    require(AIRDROP_LIMIT >= amount, 'can not airdrop more');
    AIRDROP_LIMIT = AIRDROP_LIMIT - amount;
    _mint(to, amount);
    return true;
}
```

Listing 2.10: airdrop:MobiusToken.sol

Impact The `totalSupply()` may exceed `MAX_SUPPLY`.

Suggestion N/A

2.2.3 Unfair Global Debt Mechanism

Status Not an issue, this is by design.

Description The global debt mechanism requires that all users share the platform's total debt, in another word, a user may be responsible for other users' debts. If the platform total debt (`platTotalDebt`) exceeds users total debt (`usersTotalDebt`), then the newcomers have to take the excessive debt from the beginning. Such a mechanism seems to be unfair.

For example: The platform has only a single user A, A trades 100 moUSD for 1 moTSLA. Afterwards, the price of TSLA rises to 150. The platform total debt is now 150 USD but the accounting debt from the user is only 100 moUSD. If another B wants to enter the market, he must take an extra debt proportional to `platTotalDebt` - `usersTotalDebt`.

```
function _getDebt(
    bytes32 stake,
    address account,
    bytes32 debtType
) private view returns (uint256, uint256, uint256, uint256) {
    //we should calc dynamic debt here.
    (uint256 debt, uint256 originalDebt, uint256 time) = Storage().getDebt(stake, account,
        debtType);
    if (debt == 0) {
        return (0,0,0,0);
    }
    (uint256 platTotalDebt ,uint256 usersTotalDebt ,uint256 usersTotalDebtOriginal) =
        getDynamicTotalDebt();
    uint256 dynamicDebtsTotal;
    uint256 synthPrice = AssetPrice().getPrice(debtType);
    if (platTotalDebt >= usersTotalDebt) {
        dynamicDebtsTotal = platTotalDebt - usersTotalDebt;
        return (debt + (dynamicDebtsTotal.decimalDivide(synthPrice)) * originalDebt /
            usersTotalDebtOriginal, debt, originalDebt, time);
    }
    dynamicDebtsTotal = usersTotalDebt - platTotalDebt;
    return (debt - (dynamicDebtsTotal.decimalDivide(synthPrice)) * originalDebt /
        usersTotalDebtOriginal, debt, originalDebt, time);
}
```

Listing 2.11: getDebt:Issuer.sol

Impact Users may undertake more debt than he's origin.

Suggestion N/A

2.2.4 Unclear Trading Fee Distribution Mechanism

Status Not an issue. As stated by the developers, "We are discussing with DAO about this part of logic. For now, we just store these fees in TRADING_FEE_ADDRESS".

Description The mechanism of distribution of trading fee which is collected in `Issuer` and `Trader` is unclear. Current implementation only transfers trading fee to `TRADING_FEE_ADDRESS`, and no further distribution logic (e.g. distributing fee to liquidity providers, users, governance, etc.) is provided.

Impact N/A

Suggestion N/A

2.3 Additional Recommendation

2.3.1 Code Typos

Status Acknowledged and partially fixed.

Description There are some typos in code:

- line 120 in `Trader` (`toSynthPrice` => `toSynthPirce`).
- line 179 in `RewardTrading` (`Withdraw` => `Wthdraw`).

```
119    ...
120    uint256 toSynthPirce,
121    ...
```

Listing 2.12: Trader.sol

```
178    ...
179    function Wthdraw(uint256 _pid) external override nonReentrant{
180    ...
```

Listing 2.13: RewardTrading.sol

Impact N/A

Suggestion Revise the typos.

2.3.2 Formula Inconsistency

Status Confirmed.

Description The formulas for calculating trading fee are different in code implementation, code comments and docs:

- In code implementation:

$$x = N + min\left(M \times K \times (1 + K_1 - K), M\right)$$

- In code comments and docs:

$$x = (N + M \times K) \times (1 + K_1 - K)$$

```
// assume that long position is a, short position is b, moUSD position is c,
// then net long position percentage K is |a-b|/ (a+b+c), fee rate x = N + M * K (N is basic
    fee rate,M is max fee rate offset)
// considering that newDebt position(h) affects fee rate then k1 = (|a-b|+h) / (a+b+c), x=(N +
    M * K)*(1+K1-K)
// we split the function to prevent deep stack.
function getDynamicTradingFeeRate(bytes32 synth, uint256 amountInUSD, bool isShort) external
    override view returns (uint256) {
  uint256 N = Setting().getTradingFeeRate(synth);
  if (!dynamicFee) {
    return N;
```

```
    }

    uint256 M = Setting().getMaxTradingFeeOffsetRate();
    (uint256 K,uint256 K1) = _getV(synth, amountInUSD, isShort);

    return N + ((M.decimalMultiply(K)).decimalMultiply(PreciseMath.DECIMAL_ONE() + K1 - K)).min
        (M);
}
```

Listing 2.14: getDynamicTradingFeeRate:DynamicTradingFee.sol

After confirmation, the formula in code implementation is correct, while “the formula in the white paper is flawed”.

Impact N/A

Suggestion Revise code comments & docs.

2.3.3 Redundant Function

Status Confirmed and fixed.

Description `RewardCollateralStorage` and `RewardTradingStorage` both implement `setTotalAllocPoint()`, which could only be invoked by their managers (`RewardCollateral` and `RewardTrading`, respectively). However, Neither `RewardCollateral` nor `RewardTrading` uses `setTotalAllocPoint()`. In another word, `setTotalAllocPoint()` will **NEVER** be invoked.

```
function setTotalAllocPoint(uint256 v) external override onlyManager(managerName) returns (
    bool){
    totalAllocPoint = v;
    return true;
}
```

Listing 2.15: setTotalAllocPoint

Impact Redundant function could waste gas when deploying contracts.

Suggestion Delete `setTotalAllocPoint()`.

2.3.4 Do Not Use Elastic Supply Tokens

Status Not an issue. The developers guarantee that the rebase model will not be used.

Description & Suggestion Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. Such as inflationary token, deflationary token, rebasing token, and so forth. Such a mechanism makes a DeFi system over complex. For example, a DEX using deflationary token must double check the token transfer amount when taking swap action because of the difference of actual transfer amount and parameter. The abuse of elastic supply tokens will make the DeFi system vulnerable. In reality, many security accidents are caused by the elastic supply tokens. In terms of confidentiality, integrity and availability, we highly recommend that do not use elastic supply tokens.

Impact N/A

Suggestion N/A

2.4 Other Concern

We list some of our concerns from the security perspective of the whole ecosystem, which deserve a further investigation for better mitigations.

2.4.1 Potential Front-running Due to the Price Oracle

Generally, the design and implementation of the price oracle may be abused to launch front-running. For example, the attacker could monitor the pending transactions about ChainLink, and insert an order-making transaction before price update transactions. As a MEV issue that affects the entire blockchain protocol markets, users may seek some advanced solutions to mitigate it.

Chapter 3 Conclusion

In this audit, we have analyzed the business logic, the design, and the implementation of the Mobius Contracts. Indeed, we are impressed by the design of Mobius Contracts that tries to provide flash minting services with a decentralized solution. Overall, the current code base is well structured and implemented.

Meanwhile, as previously disclaimed, this report does not give any warranties on discovering all security issues of the smart contracts. We appreciate any constructive feedback or suggestions.