KEKKAI

MNER

Smart Contract Security Audit

Audited by

KEKKAI

Supported by

BEOSIN Blockchain Security

No.202403181400

Mar 18th, 2024

Contents

Summary of Audit Results

After auditing, 1 High-risk, 1 Medium-risk, 1 Low-risk and 2 info item were identified in the MNER project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

High	Fixed: 1	Acknowledged: 0
Medium	Fixed: 1	Acknowledged: 0
Low	Fixed: 1	Acknowledged: 0
Info	Fixed: 2	Acknowledged: 0

Notes:

- Do not make the signature address of the MNERSale contract the same as the signature address of the MineralGo contract, as this may lead to signature reuse.
- When the project is officially launched, please ensure that the project parameters are correctly initialized.
- This audit only covers the contract part and does not include the security of the signature server. Please ensure the project team pays attention to the security of the signature server.

- **Project Description:**

1. Basic Token Information

Token name	MNER
Token symbol	MNER
Decimals	18
Pre supply	2,100,000,000
Total Supply	2,100,000,000 (The total supply is constant)
Token type	ERC-20

Table 1 MNER token info

2. Business overview

The MNER project is composed of multiple components, including ERC20 contracts, ERC721 contracts, token presale contracts, and staking mining contracts. Among these, the MNERSale contract serves as the core of the project, primarily facilitating the presale of tokens and supporting multiple rounds of presale activities, enabling users to participate in token presales with ETH. On the other hand, the MineralGo contract allows users to stake specific tokens and redeem corresponding rewards based on the obtained signatures. The locking cycle of these token is 1 month, half a year, and 1 year, providing users with different options for token locking durations.

1 Overview

1.1 Project Overview

Project Name	MNER
Project Language	Solidity
Platform	Ethereum
Sha256	**39AF122BFF34B4C02774F644FE00C05E652BE0FB351BBC9C7A7BD14BBE3735BE**

1.2 Audit Overview

Audit work duration: Feb 14, 2024 – Mar 18, 2024
Audit team: KEKKAI Security Team

1.3 Audit Method

The audit methods are as follows:

-Formal Verification
Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

-Manual Review
Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:
The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

2 Findings

Index	Risk description	Severity level	Status
MNER-01	Incomplete signature verification	High	Fixed
MNER-02	Collateral tokens are at risk of being lost	Medium	Fixed
MNER-03	Risk of re-entry	Low	Fixed
MNER-04	Lack of event triggering	Info	Fixed
MNER-05	Redundant code	Info	Fixed

Finding Details:

[MNER-01] Incomplete signature verification

Severity Level	High
Type	Business Security
Lines	MineralGo.sol# 318-364
Description	In the MineralGo contract, there are two withdrawal functions: `claimMNER` and `claimETH`. Among them, the `claimMNER` function is specifically designed for withdrawing MNER tokens, while the `claimETH` function is used for withdrawing ETH. However, since the signature content does not clearly indicate the type of token being withdrawn, it may result in users mistakenly using the signature intended for withdrawing MNER tokens to attempt withdrawing ETH.

```
   function claimMNER(
        uint256 claimId,
        address user,
        uint256 amount,
        uint256 deadline,
        bytes memory signature
    ) external {
     ... ...
        require(
            verifySign(claimId, user, amount, deadline, signature),
            "Invalid signature"
        );
        claimes[claimId] = true;
        IERC20(awardToken).transfer(user, amount);
        emit Claim(claimId, user, amount, deadline,
manager,block.timestamp);
    }
    function claimETH(
        uint256 claimId,
        address user,
        uint256 amount,
        uint256 deadline,
        bytes memory signature
```

```
    ) external {
    ... ...
        require(
            verifySign(claimId, user, amount, deadline, signature),
            "Invalid signature"
        );
        payable(user).transfer(amount);
        claimes[claimId] = true;
        emit ClaimETH(claimId, user, amount, deadline,
manager,block.timestamp);
    }
```

Recommendation	In MineralGo contracts, in order to ensure the correctness and security of function calls, it is recommended that the signature content explicitly include the type of tokens received.
Status	**Fixed.**

```
            function claim(
        uint256 claimId,
        uint256 claimType,
        address user,
        uint256 amount,
        uint256 deadline,
        bytes memory signature
    ) external {
    ... ...
        require(
            verifySign(claimId, claimType,user, amount, deadline,
signature),
            "Invalid signature"
        );
        claimes[claimId] = true;
        if(claimType == 0) {
            IERC20(awardToken).transfer(user, amount);
        } else {
            payable(user).transfer(amount);
        }
        emit Claim(claimId, claimType, user, amount, deadline, manager,
block.timestamp);
    }
```

[MNER-02] Collateral tokens are at risk of being lost

Severity Level	**Medium**
Type	Business Security
Lines	MineralGo.sol# 402-409
Description	In the MineralGo contract, as users' collateral is stored within the contract, the leakage of the contract owner's private key poses a significant security risk. The leakage of the private key allows attackers to impersonate the contract owner and execute any contract function, including the transfer of users' collateral. In such a scenario, users' collateral may be lost or illegally transferred, leading to the compromise of their assets.
Recommendation	It is recommended to remove the `withdrawTokensSelf` function or use a multi-signature wallet to manage the owner permissions.
Status	**Fixed.** This function has been deleted in the code.

```
        function withdrawTokensSelf(address token, address to) external
onlyOwner {
        if (token == address(0)) {
            payable(to).transfer(address(this).balance);
        } else {
            uint256 bal = IERC20(token).balanceOf(address(this));
            IERC20(token).transfer(to, bal);
        }
    }
```

[MNER-03] Risk of re-entry

Severity Level	**Low**
Type	General Vulnerability
Lines	MineralGo.sol#219-237
Description	In the `redeemMNER` function, the setting of collateral tokens relies on the contract owner. If ERC20 tokens that support callbacks are introduced, it may increase the risk of re-entrancy attacks. This is because ledger updates occur after the transfer operation.

```
        function redeemMNER(uint _orderId) public {
        require(
            morders[_orderId].user == msg.sender,
            "Mineral: Order is incorrect"
        );
        require(
            morders[_orderId].redeem != true,
            "Mineral: Order has been redeemed"
        );
        require(
            block.timestamp > morders[_orderId].unlockTime,
            "Mineral: Order has not yet reached the redemption time"
        );
        IERC20(morders[_orderId].token).transfer(msg.sender,
morders[_orderId].amount);
        morders[_orderId].redeem = true;
        emit RedeemMNER(msg.sender, _orderId, block.timestamp);
    }
```

Recommendation	It is recommended that projects consider the risk of re-entry, and it is recommended to use the openzepplin anti-re-entry modifier or to put the book update in front of the transfer.
Status	**Fixed.**

```
     function redeemMNER(uint _orderId) public {
        require(
            morders[_orderId].user == msg.sender,
            "Mineral: Order is incorrect"
        );
        require(
```

```
            morders[_orderId].redeem != true,
            "Mineral: Order has been redeemed"
        );
        require(
            block.timestamp > morders[_orderId].unlockTime,
            "Mineral: Order has not yet reached the redemption time"
        );
        morders[_orderId].redeem = true;
        IERC20(morders[_orderId].token).transfer(msg.sender,
morders[_orderId].amount);
        emit RedeemMNER(msg.sender, _orderId, block.timestamp);
    }
```

[MNER-04] Lack of event triggering

Severity Level	Info
Type	Coding Conventions
Lines	MineralGo.sol,MNERSale.sol#176-193,72-83
Description	In the MineralGo and MNERSale contracts, there is a lack of event logging for the following functions. In blockchain smart contracts, events are a crucial mechanism that allows contracts to emit notifications when specific conditions are met, leaving tamper-proof records on the blockchain. These events are essential for contract transparency, auditability, and interaction with external listeners such as frontend applications or data analysis tools.

```
    function setMineralToken(uint256 tokenType, address _token)
external onlyOwner {

        Mineral[tokenType] = _token;
    }
    function setMNERToken(uint256 tokenType, address _token) external
onlyOwner {

        MNER[tokenType] = _token;
    }
    function setAwardToken(address _token) external onlyOwner {
        awardToken = _token;
    }
    function setMProduct(uint id, uint _times) external onlyOwner {
        mproduct[id] = _times;
    }
    function setProduct(uint id, uint _times) external onlyOwner {
        product[id] = _times;
    }
    function setTreasuryWallet(address _wallet) external onlyOwner {
        treasuryWallet = _wallet;
    }
     function setTime(uint256 _startTime,uint256 _endTime) external
onlyOwner {
        startTime = _startTime;
```

	`` endTime = _endTime; } ```
Recommendation	Event triggering is recommended for the above functions.
Status	**Fixed.**

```
function setMineralToken(uint256 tokenType, address _token)
        external
        onlyOwner
    {
        require(
            Mineral[tokenType] == address(0),
            "Mineral: Cannot change token address"
        );
        Mineral[tokenType] = _token;
        emit UpdateMineralToken(tokenType, _token);
    }
    function setMNERToken(uint256 tokenType, address _token)
        external
        onlyOwner
    {
        require(
            MNER[tokenType] == address(0),
            "Mineral: Cannot change token address"
        );
        MNER[tokenType] = _token;
        emit UpdateMNERToken(tokenType, _token);
    }
    function setMProduct(uint256 id, uint256 _times) external onlyOwner
{
        mproduct[id] = _times;
```

```
        emit UpdateMNERProduct(id, _times);

    }

    function setProduct(uint256 id, uint256 _times) external onlyOwner
{

        product[id] = _times;

        emit UpdateMineralProduct(id, _times);

    }

    function updateManager(address _m) external onlyOwner {

        manager = _m;

        emit UpdateManager(manager, _m);
```

[MNER-05] Redundant code

Severity Level	Info
Type	Coding Conventions
Lines	MNERSale.sol#100-114
Description	The `_safeTransferFrom` function is not used in the MNERSale contract and it is recommended that it be removed.

```
function _safeTransferFrom(
    address token,
    address from,
    address to,
    uint256 amount
) private {
    if (amount == 0) {
        return;
    }
    if (token == address(0)) {
        require(msg.value == amount);
    } else {
        IERC20(token).safeTransferFrom(from, to, amount);
    }
}
```

Recommendation	It is recommended that this function be removed.
Status	**Fixed.**

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Impact / Likelihood	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact vulnerabilities are those that can significantly compromise the confidentiality, integrity, and availability of smart contracts or their economic frameworks. These vulnerabilities can lead to major financial losses within the contract business system, extensive data breaches, loss of administrative control, failure of crucial functionalities, erosion of trust, or even indirectly impact other connected smart contracts, leading to extensive damages. Such vulnerabilities often result in severe and, in many cases, irreversible harm.

- **High**

Vulnerabilities with a high impact are those that can cause considerable damage to the confidentiality, integrity, and availability of smart contracts or their economic models. These can lead to substantial economic losses, localized dysfunctionality, erosion of trust, and other significant detrimental effects on the contract business system.

- **Medium**

Medium impact vulnerabilities are those that can moderately affect the confidentiality, integrity, and availability of smart contracts or their economic models. These vulnerabilities might lead to minor financial losses, affect individual business operations, and have other moderate repercussions on the contract business system.

- **Low**

Low impact vulnerabilities are those that marginally affect smart contracts. These vulnerabilities may pose certain security risks to the contract business system and necessitate improvements, but they generally lead to lesser concerns compared to higher-level vulnerabilities.

3.1.4 Likelihood of Exploitation

- **Probable**

A probable likelihood indicates that exploiting the vulnerability requires minimal effort or resources, without any specific conditions needing to be met. The vulnerability can be exploited reliably and consistently.

- **Possible**

A possible likelihood suggests that exploiting the vulnerability involves some level of cost or meets certain prerequisites. The vulnerability may not be triggered easily or consistently.

- **Unlikely**

An unlikely likelihood implies that exploiting the vulnerability demands considerable resources or meets stringent conditions, making it notably challenging to exploit.

- **Rare**

A rare likelihood signifies that triggering the vulnerability requires an exceptionally high level of resources or meets conditions that are extremely difficult to fulfill, making exploitation highly improbable.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

KEKKAI categorizes smart contract security issues into three distinct groups: Coding Conventions, General Vulnerability, and Business Security. Here's an overview of each category:

- **Coding Conventions**

This category assesses whether smart contracts adhere to established security coding guidelines specific to their programming language. For instance, contracts written in Solidity should lock in a specific compiler version and avoid using outdated or deprecated language features.

- **General Vulnerability**

General Vulnerability encompasses common security flaws that could be present in smart contract implementations. These vulnerabilities are typically intrinsic to the smart contract's nature and include issues like integer overflow/underflow or susceptibility to denial of service (DoS) attacks.

- **Business Security**

Business Security pertains to potential vulnerabilities directly linked to the unique business logic of each project, often exhibiting a higher degree of specificity. Examples include discrepancies between the code's lock-up terms and those outlined in the project's white paper or vulnerabilities like flash loan attacks stemming from incorrect oracle settings for price retrieval.

*It's important to note that projects may face risks associated with third-party protocols integrated into their system, which falls outside KEKKAI's purview. Business Security necessitates active involvement from the project team, who, along with their users, should remain vigilant at all times to mitigate risks effectively.

3.3 Disclaimer

The Audit Report produced by KEKKAI pertains specifically to the services outlined in the corresponding service agreement. It is intended for use solely by the Party receiving the service (hereafter referred to as the "Served Party") within the parameters and limitations set forth in the service agreement. It is not permissible for any third parties to transmit, divulge, cite, depend upon, or alter the Audit Report for any purpose.

KEKKAI's Audit Report is strictly focused on the code and should not be construed as an endorsement or validation of the project itself. It does not provide any assurances or guarantees regarding the complete integrity of the analyzed code, the team responsible for the code, the business model, or compliance with legal standards.

The insights provided in KEKKAI's Audit Report are based solely on the code as submitted by the Served Party and the technological capabilities accessible to KEKKAI at the time of the audit. However, due to inherent technological limitations and potential issues such as incomplete information, tampering, deletion, concealment, or subsequent modifications of the code by the Served Party, the report may not be able to fully identify all potential risks.

KEKKAI's Audit Report is not intended to offer investment guidance or recommendations for any project. It is designed as a thorough examination aimed at assisting our clients in enhancing the quality of their code and reducing the inherent risks associated with blockchain technology.

3.4 About KEKKAI

KEKKAI provides a web3.0 anti-fraud security solution for the consumer side based in Japan. It now offers a product range that includes an anti-fraud browser extension and a mobile application, and solution for web3.0 security such as smart contract security audit and penetration test. The aim of KEKKAI is to build the security layer of Web3 for consumers. It provides not only a firewall for daily crypto trading but also an environment where users can browse the Web3 world with peace of mind. Since its launch in February 2023, KEKKAI has gained over 40,000 users all over the world using KEKKAI's product. It is now protecting more than $200M of user asset from not being attacked, and many of projects for security auditing.

KEKKAI

0xKEKKAI

https://kekkai.io