Security audit

Mintera

2nd security review
Date : June 5st, 2023

Summary

I. Introduction

1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

2. Methodology

a. Preparation

This smart contract is a staking contract for Mintera tokens.
A first technical meeting was held on 12/05/2023. It allows technical teams to explain the contract workflow, to define the exact scope, and start the audit process.

b. Review

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- Money-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead Of Transfer
- Costly Loop
- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report which contains for each vulnerability :

- Explanation
- Severity score
- How to fix it / Recommendation

Here are severity score definitions.

Critical	A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.
Major	A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.
Medium	A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.
Low	A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.
Informational	An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

The scope of the audit is one smart contract which is stored in Mintera Github as StakingVault.sol (commit: 29b270a5dcdbabcddb96cc5faf916375c6bf90cc).

The 2nd review was done on commit 04b0f36a2ebcc25849f2f394d187b227ad95328b.

We assume that the following smart contracts don't need to be audited :
- Context.sol
- ERC20.sol
- ERC20Burnable.sol
- IERC20.sol
- IERC20Metadata.sol
- Ownable.sol
- Pausable.sol
- ReentrancyGuard.sol
- SafeMath.sol

II. Vulnerabilities

Critical

2 critical severity issues were found :

- CRIT-1 Permanent loss of USDT
- CRIT-2 Unexpected reward distribution

Major

1 major severity issue was found :

- MAJ-1 Front-running benefits

Medium

2 medium severity issues were found :

- MED-1 Centralization risks in owner role
- MED-2 Denial of Service

Low

3 low severity issues were found :

- LOW-1 Unadapted use of ERC20 standard
- LOW-2 Unnecessary code complexity
- LOW-3 Floating pragma version

Informational

10 informational severity issues were found :

- INF-1 Too many digit
- INF-2 MINIMUM_CONTRIBUTION_AMOUNT should be constant
- INF-3 Useless default boolean affectation
- INF-4 Useless boolean comparisons
- INF-4 Could check address validity
- INF-5 Inconsistent comment
- INF-6 Useless balance check
- INF-7 Unnecessary code complexity
- INF-8 Code repetition
- INF-9 Typo error in a comment
- INF-10 tokenMNTE and tokenUSDT should be immutable

CRIT-1 Permanent loss of USDT

Impact: Critical

Description:

The reward is calculated with `calculateReward` function which uses `lastRewardAmount` global variable.

```
return ((stakers[_investorAddress].stakedAmount).mul(lastRewardAmount)).div(totalStakedAmount);
```

After a deposit, the `lastRewardAmount` variable is updated. There is no check that all rewards were distributed. If there are not, some USDT can be lost on the contract address forever. Because there is no other way to send it.
Moreover, if the farmer calls the `deposit` function two times in a row by mistake, the first USDT sent will be lost.

Recommendation:

There are 3 possibilities to fix this:

1. Add a check at the beginning of the `deposit` function that verifies all rewards were sent.
2. Change the way the reward is calculated.
3. Add an emergency function that allows `farmingAddress` address to withdraw USDT from the contract.

Status:

`currentNumberOfStakersRewarded` was added in order to ensure every staker was rewarded.
The withdrawal is impossible since it is not a claiming period. To toggle claiming period, it verifies that `currentNumberOfStakersRewarded` is equal to the number of stakers.
If there is no reward, or if stakingPerriod is toggled by mistake, it can be mitigated by sending 0 USDT as a deposit and reward users.

CRIT-2 Unexpected reward distribution

Impact: Critical

Description:

The reward is calculated with `calculateReward` function which uses `lastRewardAmount` global variable.

```
return ((stakers[_investorAddress].stakedAmount).mul(lastRewardAmount)).div(totalStakedAmount);
```

After a deposit, the `lastRewardAmount` variable is updated with `_amount` value.

```
        require(
            tokenUSDT.transferFrom(_msgSender(), address(this), _amount),
            "Failed to deposit USDT tokens to the staking contract"
        );

        // Finally, we update the total reaward amount
        lastRewardAmount = _amount;
```

However, `_amount` value is not necessarily the USDT balance of this smart contract. On USDT smart contract, at address 0xdac17f958d2ee523a2206206994597c13d831ec7, a fee can be taken from Tether organization. In the case `fee` is higher than 0, `lastRewardAmount` will be higher than the smart contract balance. It can lead to unexpected reward distribution.

Recommendation:

Use `tokenUSDT.balanceOf(address(this))` to update `lastRewardAmount`. Warning: it may have impacts on other pieces of code.

Else, you can create a function to calculate USDT fees. You need to call the USDT contract to get `basisPointsRate` and `maximumFee` variables.

```
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
```

Status:

An error was created with this change in actual smart contract:

```
// Then, we transfer reward tokens of sender to this contract
require(
    tokenUSDT.transferFrom(_msgSender(), address(this), usdtAmount),
    "Failed to deposit USDT tokens to the staking contract"
);

// Finally, we update the total reward amount
lastRewardAmount = _amount;
```

Need to be replaced by:

```
// Then, we transfer reward tokens of sender to this contract
require(
    tokenUSDT.transferFrom(_msgSender(), address(this), _amount),
    "Failed to deposit USDT tokens to the staking contract"
);

// Finally, we update the total reward amount
lastRewardAmount = usdtAmount;
```

Else, the fix will work.

Finally, the smart contract will be added to the Arbitrum blockchain and not on Ethereum. Because the USDT smart contract on Arbitrum is not the same, fees are not possible in the actual smart contract.
Nevertheless, USDT smart contract works with a proxy, so it is not immutable.

Update 5/06 : It was fixed.

MAJ-1 Front-running benefits

Impact: Major

Description:

`farmingAddress` address is allowed to deposit only before the claiming period. There is no advantage to stake before a long before claiming period.

A malicious actor could start staking just before the `toggleClaimPeriod` function call by the owner with a very big percentage of MNTE staked. In that way, he will be able to win as much USDT from staking reward as a user who is staking for a longer period.

Recommendation:

To avoid this point of vulnerability, the best solution should be to calculate reward for each user when deposit depending on stake timestamp. This involves important changes in the smart contract workflow.

An easy fix could be to replace `isClaimPeriod` bool and use an `Enum` type with 3 values:

```
enum Periods {
    Staking,
    Waiting,
    Claiming
}
```

A waiting time between staking and claiming could reduce front-run impact.

Then an additional modifier and function need to be coded:

- a modifier to check whether an external call is authorize depending on `Periods`
- a function to allow owner to change period

This fix does not completely resolve fair reward distribution. We highly encourage to calculate rewards depending on timestamp (or block number).

Status:

Those 3 periods were added.
The logic was changed, allowing users to withdraw in the second claiming period after they stack.
A new issue appears when a user withdraw. He can't claim the last reward he should have one month after he unstaked because he was rejected from `stakerAddressList`. Moreover, it leads to USDT lost on the contract. Here are 2 solutions:

1/ Make some changes in the code that will increase complexity:
- adding a trigger on stakers who decided to `_unstake`.
- realize _unstake operations after their `yieldRewardAmount` was updated.

2/ Accept that users can't claim the last month. Then :
- add a new variable `rewardOverflow` that saves last month's rewards.
- create a function that allows farmer to withdraw those rewards

Even if this is an unwanted behavior, we strongly advise to solve this issue with the second point. A v2 would be better to don't add too much complexity on the contract design.
It was fixed in the last update, allowing the farmer to withdraw overflow USDT.

Update 5/06 : The second solution was implemented.

MED-1 Centralization risks in owner role

Impact: Medium

Description:

The owner role can be assigned to a single externally owned account (EOA). It can lead to centralization and an increased risk of private key leaks.

Recommendation:

To mitigate this risk, we recommend using a multisignature wallet that is jointly owned by multiple individuals. This would distribute control and reduce the likelihood of a single point of failure.

Status:

A multisig wallet will be used on mainnet deployment.

MED-2 Denial of Service

Impact: Medium

Description:

In the reward calculation, `stakedAmount` and `lastRewardAmount` are multiplicated before being divided.

```
return ((stakers[_investorAddress].stakedAmount).mul(lastRewardAmount)).div(totalStakedAmount);
```

If an overflow happens, it can lead to a DoS on `calculateReward` function and so on `distribute` function.

While MNTE can't be minted and maximum supply is 65e24, the probability is not null: USDT can be minted without any limits. However, today's USDT total supply is 36283188702721368.

Recommendation:

Add an emergency function that allows `farmingAddress` address to withdraw USDT from the contract. In that way, if it happens, no USDT will be lost.

Status:

Seeing the MNTE and USDT maximum supply, it is now impossible. The Mintera team assumes that USDT supply will not change in this magnitude, which is understandable.

LOW-1 Unadapted use of ERC20 standard

Impact: Low

Description:

To send MNTE tokens from the contract itself, `transferFrom` function is used. It adds complexity and gas without advantages of a traditional `transfer`.

```
tokenMNTE.approve(address(this), amount);
require(
    tokenMNTE.transferFrom(address(this), _msgSender(), amount),
    "Failed to withdraw MNTE tokens from the staking contract"
);
```

Recommendation:

Replace the lines above by:

```
require(
    tokenMNTE.transfer(_msgSender(), amount),
    "Failed to withdraw MNTE tokens from the staking contract"
);
```

If this fix is not done, then the boolean returned by `approve` should be check with the following line:

```
require(tokenMNTE.approve(address(this), amount));
```

Status:

The fix is implemented, following recommendations.

LOW-2 Unnecessary code complexity

Impact: Low

Description:

`address(0)` is added to `stakerAddressList` in the constructor. It was done to bypass an if statement for the first staker line 122.

```
// By default we set the null address in the array
// We will use the index 0 in order to check if an investor has been referenced in the array
stakerAddressList.push(address(0));
```

However, it is not a good idea because :
1/ it is misunderstanding for an external user
2/ if an undetected issue leads to remove a null address from the list, it becomes critical
3/ `numberOfStakers` function is then false
4/ following `require` is always `True` then, which is not desired

```
require( stakerAddressList.length > 0, "No investors in the list of addresses");
```

Recommendation:

There are other ways to bypass the if statement line 122 :

```
if (stakers[_msgSender()].stakerIndex == 0) {
```

It can be done by checking `stakedAmount` or `lastStakeTimestamp`.

```
if (stakers[_msgSender()].lastStakeTimestamp == 0) {
```

Then, we recommend removing the push of `address(0)` in the constructor.

Status:

The fix is implemented, following recommendations.

LOW-3 Floating pragma version

Impact: Low

Description:

The version of Solidity is not fixed. Consider locking the version pragma to the same Solidity version used during development and testing. Also consider setting this version to be the latest release.

Recommendation:

It is always recommended that pragma should be fixed to the version that you are intending to deploy your contracts with. Replace the floating pragma version (line 2) by:

```
pragma solidity 0.8.17;
```

Status:

The fix is implemented, following recommendations.

INF-1 Too many digit

Impact: Informational

Description:

`MINIMUM_CONTRIBUTION_AMOUNT` is hard to read because of too many digits. It's easy to misread the number of zeros in big numbers.

```
// Minimum Amount to Stake in MNTE
uint256 public MINIMUM_CONTRIBUTION_AMOUNT = 25000000000000000000;
```

Recommendation:

We recommend using scientific notation.

```
// Minimum Amount to Stake in MNTE
uint256 public MINIMUM_CONTRIBUTION_AMOUNT = 25e20;
```

Status:

The fix is implemented, following recommendations.

INF-2 `MINIMUM_CONTRIBUTION_AMOUNT` should be constant

Impact: Informational

Description:

`MINIMUM_CONTRIBUTION_AMOUNT` variable can't be modified. It should be constant. In addition optimizing gas, the naming convention will be respected.

```
// Minimum Amount to Stake in MNTE
uint256 public MINIMUM_CONTRIBUTION_AMOUNT = 25000000000000000000;
```

Recommendation:

We recommend using the `constant` keyword.

```
// Minimum Amount to Stake in MNTE
uint256 public constant MINIMUM_CONTRIBUTION_AMOUNT = 25000000000000000000;
```

Status:

The fix is implemented, following recommendations.

INF-3 Useless default boolean affectation

Impact: Informational

Description:

`isClaimPeriod` variable is a bool. All boolean variables have a `false` default value. There is no need to set it to false in the constructor.

```
        // By default, we set to the staking period
        isClaimPeriod = false; // By default
```

Recommendation:

Delete the code above (lines 95-96).
You can also write a comment next to `isClaimPeriod` variable declaration.

```
    // True if it is a claim period, else it is an investment period
    bool public isClaimPeriod; //False by default
```

Status:

This part has been fixed with MAJ-1 changes.

INF-4 Useless boolean comparisons

Impact: Informational

Description:

Booleans should not be compared to `False` or `True`. It is gas consuming and adds complexity.

Recommendation:

Here are the lines that can be modified:

- Lines 110-113:

```
        require(
            isClaimPeriod == false,
            "Staking period has not been reached yet"
        );
```

should be replaced by:

```
        require(
            !isClaimPeriod,
            "Staking period has not been reached yet"
        );
```

- Lines 180-183:

```
        require(
            isClaimPeriod == true,
            "Claiming period has not been reached yet"
        );
```

should be replaced by:

```
        require(
            isClaimPeriod,
            "Claiming period has not been reached yet"
        );
```

- Lines 200-203:

```
require(
    isClaimPeriod == true,
    "Claiming period has not been reached yet"
);
```

should be replaced by:

```
require(
    isClaimPeriod,
    "Claiming period has not been reached yet"
);
```

- Lines 259-262:

```
require(
    isClaimPeriod == false,
    "Staking period has not been reached yet"
);
```

should be replaced by:

```
require(
    !isClaimPeriod,
    "Staking period has not been reached yet"
);
```

Status:

This part has been fixed with MAJ-1 changes.

INF-4 Could check address validity

Impact: Informational

Description:

In the transferFarming function, _newFarmer variable is not checked. Impact is very low since the `owner` can modify farmingAddress again.

```
function transferFarming(address _newFarmer) public onlyOwner {
    farmingAddress = _newFarmer;
}
```

Recommendation:

Add the `require` as the following code:

```
function transferFarming(address _newFarmer) public onlyOwner {
    require(_newFarmer != address(0), "New farmer is the zero address");
    farmingAddress = _newFarmer;
}
```

The same logic can be implemented in the constructor `farmingAddress` check.

Status:

The fix is implemented, following recommendations.

INF-5 Inconsistent comment

Impact: Informational

Description:

In the following lines, the comments are not relevant.

```
        // We first transfer staked tokens of sender to this contract
        // tokenMNTE.approve(address(this), _amount);
```

Recommendation:

Remove those comments. A comment line could be added at the beginning of the function to warn users to give allowance first.

```
    /**
     * @dev Allows investor to stake his MNTE tokens in the contract
     * @notice Contract must not be paused to run this function
     * @notice Minimum staking amount: 2500 MNTE
     * @notice The investor must have at least the amount sent as parameter in his MNTE balance
     * @notice The investor must give a minimum allowance of _amount to this contract
     * @notice The contract must be set in a claiming period
     * @param _amount {{uint256}} - Amount of MNTE tokens going to be staked
     */
```

Status:

The fix is implemented, following recommendations.

INF-6 Useless balance check

Impact: Informational

Description:

There is no need to check the balance of the sender. It is done when calling `transferFrom` ERC20 function.

```
require(tokenMNTE.balanceOf(_msgSender()) >= _amount, "Insufficient Balance of MNTE to stake");
```

Recommendation:

We recommend removing line 109 to save gas and increase readability.

Status:

The fix is implemented, following recommendations.

INF-7 Unnecessary code complexity

Impact: Informational

Description:

In the `calculateReward` function, there is no utility to check if `stakedAmount > 0`. At the end, if `stakedAmount == 0`, the function will return 0.

```
    function calculateReward(address _investorAddress) public view returns (uint256) {
        if (
            stakers[_investorAddress].lastRewardTimestamp < lastRewardTimestamp &&
            stakers[_investorAddress].stakedAmount > 0
        ) {
            // In the case investor last reward timestamp is less than last deposit of USDT
            // AND if the investor is staking at least one token
            return
((stakers[_investorAddress].stakedAmount).mul(lastRewardAmount)).div(totalStakedAmount);
        } else {
            return 0;
        }
    }
```

Recommendation:

We recommend removing this check in order to win a lot of readability, and save some gas.

```
    function calculateReward(address _investorAddress) public view returns (uint256) {
        if (
            stakers[_investorAddress].lastRewardTimestamp < lastRewardTimestamp
        ) {
            // In the case investor last reward timestamp is less than last deposit of USDT
            return ((stakers[_investorAddress].stakedAmount).mul(lastRewardAmount)).div(totalStakedAmount);
        } else {
            return 0;
        }
    }
```

Status:

The logic has changed with MAJ-1 modifications.

INF-8 Code repetition

Impact: Informational

Description:

In the `stake` function, code can be simplified. Some lines are redundant. There is no need to separate cases when it is a new staker or not:

- to increase `stakedAmount`
- to update `lastStakeTimestamp`

```
if (stakers[_msgSender()].stakerIndex == 0) {
    // Then we add its address to the investor list
    stakerAddressList.push(_msgSender());

    // In this case the Staker struct has still not been defined yet
    Staker memory staker;
    staker.stakedAmount = _amount;
    staker.lastStakeTimestamp = block.timestamp;
    staker.stakerIndex = stakerAddressList.length - 1;

    // Finally we add it to the mapping
    stakers[_msgSender()] = staker;
} else {
    // In the case msg.sender has already staked before
    stakers[_msgSender()].stakedAmount = stakers[_msgSender()].stakedAmount.add(_amount);
    stakers[_msgSender()].lastStakeTimestamp = block.timestamp;
}
```

Recommendation:

We recommend replacing code by those lines in order to win readability and save gas.

```
if (stakers[_msgSender()].stakerIndex == 0) {
    // Then we add its address to the investor list
    stakerAddressList.push(_msgSender());

    // In this case the Staker struct has still not been defined yet
    Staker memory staker;
    staker.stakerIndex = stakerAddressList.length - 1;

    // Add it to the mapping
    stakers[_msgSender()] = staker;
}
    stakers[_msgSender()].stakedAmount = stakers[_msgSender()].stakedAmount.add(_amount);
    stakers[_msgSender()].lastStakeTimestamp = block.timestamp;
```

Status:

The fix is implemented, with a difference caused by the new `firstStakeTimestamp` variable.

INF-9 Typo error in a comment

Impact: Informational

Description:

There is a small typo in a comment.

```
// Finally, we update the total reaward amount
```

Recommendation:

It can easily be fixed.

```
// Finally, we update the total reward amount
```

Status:

The fix is implemented, following recommendations.

INF-10 `tokenMNTE` and `tokenUSDT` should be immutable

Impact: Informational

Description:

`tokenMNTE` and `tokenUSDT` variables can't be modified after deployment. It should be `immutable` to optimize gas.

```
// MNTE Token
IERC20 tokenMNTE;
// USDT Token
IERC20 tokenUSDT;
```

Recommendation:

We recommend using the `immutable` keyword.

```
// MNTE Token
IERC20 immutable tokenMNTE;
// USDT Token
IERC20 immutable tokenUSDT;
```

Status:

The fix is implemented, following recommendations.