METATRUST

Security Assessment for

# Bitmap

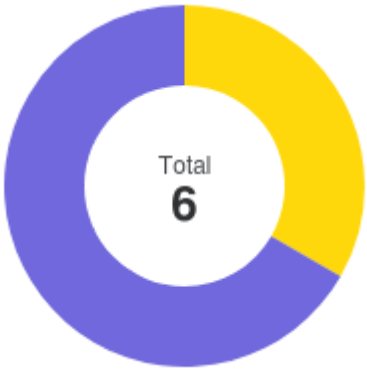
August 20, 2024

## Executive Summary

| Overview | |
|---|---|
| Project Name | Bitmap |
| Codebase URL | https://github.com/bitmap-game/bitmap-contracts/tree/main |
| Scan Engine | Security Analyzer |
| Scan Time | 2024/08/20 08:00:00 |
| Commit Id | c9217ecac6e6503427559578464c4e791776fe3add9d38162390876b3a30e3ffe257b9435a276fd4 |

| Total | |
|---|---|
| Critical Issues | 0 |
| High risk Issues | 0 |
| Medium risk Issues | 2 |
| Low risk Issues | 4 |
| Informational Issues | 0 |

| | |
|---|---|
| Critical Issues | The issue can cause large economic losses, large-scale data disorder, loss of control of authority management, failure of key functions, or indirectly affect the correct operation of other smart contracts interacting with it. |
| High Risk Issues | The issue puts a large number of users' sensitive information at risk or is reasonably likely to lead to catastrophic impacts on clients' reputations or serious financial implications for clients and users. |
| Medium Risk Issues | The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact. |
| Low Risk Issues | The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances. |
| Informational Issue | The issue does not pose an immediate risk but is relevant to security best practices or Defence in Depth. |




| | | |
|---|---|---|
| Critical Issues | 0% | 0 |
| High risk Issues | 0% | 0 |
| Medium risk Issues | 33% | 2 |
| Low risk Issues | 67% | 4 |
| Informational Issues | 0% | 0 |

## Summary of Findings

MetaScan security assessment was performed on **August 20, 2024 08:00:00** on project **Bitmap** with the repository on branch **default branch**. The assessment was carried out by scanning the project's codebase using the scan engine **Security Analyzer**. There are in total **6** vulnerabilities / security risks discovered during the scanning session, among which **2** medium risk vulnerabilities, **4** low risk vulnerabilities,

| ID | Description | Severity | Alleviation |
|---|---|---|---|
| MSA-001 | The `withdrawer` May Withdraw More Rent Fee Than Expected | Medium risk | Mitigated |
| MSA-002 | The Bad Debt Probably Increases As Time Goes By | Medium risk | Mitigated |
| MSA-003 | If 1% of the Deposit is Less Than the Gas Fee, None of Liquidator Will Liquidate a Rent | Low risk | Mitigated |
| MSA-004 | The Implementation of `verifyRentSignature` Missing a Few Good Practices | Low risk | Acknowledged |
| MSA-005 | The `disableInitializers` Function is Missed for Upgradeable Contracts | Low risk | Fixed |
| MSA-006 | The `getTotalReward` Does not Exclude the Bad Debt | Low risk | Mitigated |

# Findings

## Medium risk (2)

### 1. The `withdrawer` May Withdraw More Rent Fee Than Expected Medium risk Security Analyzer

The `withdrawReward` function can withdraw rent fee as much as the `rentStat.totalRentFee`, no matter the `rentStat.totalRentFee` contains the bad debt or not.

Note that the `_updateStopRentStat` will incur the rent fee based on the `rentStat.totalRentDeposit`, no matter if there are bad debt in the total rent deposit:

```
function _updateStopRentStat(uint256 rentAmount) internal {
    //firstly update totalRentFee, then totalRentDeposit, finally updateTime.
    uint256 interval = block.timestamp - rentStat.updateTimestamp;
    uint256 feePerSecond = _rentFeePerSecond(rentStat.totalRentDeposit);

    rentStat.totalRentFee += feePerSecond * interval;
    rentStat.totalRentDeposit -= rentAmount;
    rentStat.updateTimestamp = block.timestamp;
}
```

The bad result is that a normal user can not stop a rent after a withdraw by the withdrawer.

A simplied example:

- Alice starts a rent, and deposit 10000 $bitmap;
- after a long time, like 1 year, the rent fee becomes 12000 $bitmap, and no one liquidate Alice's bad debt;
- The `totalRentFee` is 12000 $bitmap;
- Bob starts a rent, and deposits 20000 $bitmap;
- One second later,
- The withdrawer withdraws 12000 $bitmap;
- Bob stops the rent failed, assume rent fee of Bob is 10 $bitmap, due to `deposit - rentFee` is 19990 $bitmap but the $bitmap balance of the contract is, (10000 $bitmap + 20000 $bitmap - 12000 $bitmap), 18000 $bitmap.

**File(s) Affected**

contracts/BitmapRent.sol #253-275

```
    function liquidateRent(string memory _rentId) external whenNotPaused nonReentrant {
        require(!rentIdToRent[_rentId].stopped, "rent already terminated");

        Rent storage rent = rentIdToRent[_rentId];

        //update stat
        _updateStopRentStat(rent.deposit);

        rent.stopped = true;
        rent.stopTimestamp = block.timestamp;
        rent.rentFee = _calRentFee(rent);

        //excessive rent fee
        if (rent.rentFee > rent.deposit) {
            rent.stoppedState = StoppedState.AbnormalLiquidated;
            uint256 badDebts = rent.rentFee-rent.deposit;

            //liquidate: repay bad debts
            IERC20(bitmapToken).transferFrom(msg.sender, address (this), badDebts);

            emit LiquidateRent(msg.sender, _rentId, StoppedState.AbnormalLiquidated, 0, badDebts);
            return;
        }
```

contracts/BitmapRent.sol #348-359

```
    function withdrawReward(uint256 _amount) external whenNotPaused nonReentrant {
        require(_amount > 0, "invalid _amount");
        require(msg.sender == withdrawer, "only stake contract allowed");

        _updateRentStat();
        require(rentStat.totalWithdrawnRentFee + _amount <= rentStat.totalRentFee, "amount exceed");
        rentStat.totalWithdrawnRentFee += _amount;

        IERC20(bitmapToken).transfer(withdrawer, _amount);

        emit WithdrawReward(msg.sender, _amount);
    }
```

**Recommendation**

Consider excluding the bad debt when withdrawing the reward by the withdrawer.

**Alleviation** Mitigated

The team reply: The daily fee is one ten-thousandth, in theory, most of the settlement logic is processed before the end of the lease, so there will be no problem. It will only occur after the lease period has expired, and the probability is very small. The amount is about (one ten-thousandth of the total amount / number of valid leases) of the lease contract amount, which can be almost ignored. Bad debts are settled by the official guarantee.

## 2. The Bad Debt Probably Increases As Time Goes By

Medium risk Security Analyzer

If there is a rent that has more rent fee than its deposit, it can not be stopped and can only be liquidated. At this case, the `liquidateRent` requires liquidator to pay the bad debt without any benifit:

```
    function liquidateRent(string memory _rentId) external whenNotPaused nonReentrant {
        ...
            uint256 badDebts = rent.rentFee-rent.deposit;
```

```
            //liquidate: repay bad debts
            IERC20(bitmapToken).transferFrom(msg.sender, address (this), badDebts);
```

Thus, it is probably none of liquidators will prefer to liquidate a bad debt.

On the other side, the bad debt increases as time goes by.

```
rentFee += baseRentFee + feePerSecond * (end - begin);
```

Finally, the bad debt is probably increases all the time.

**File(s) Affected**

contracts/BitmapRent.sol #253-275

```
253     function liquidateRent(string memory _rentId) external whenNotPaused nonReentrant {
254         require(!rentIdToRent[_rentId].stopped, "rent already terminated");
255
256         Rent storage rent = rentIdToRent[_rentId];
257
258         //update stat
259         _updateStopRentStat(rent.deposit);
260
261         rent.stopped = true;
262         rent.stopTimestamp = block.timestamp;
263         rent.rentFee = _calRentFee(rent);
264
265         //excessive rent fee
266         if (rent.rentFee > rent.deposit) {
267             rent.stoppedState = StoppedState.AbnormalLiquidated;
268             uint256 badDebts = rent.rentFee-rent.deposit;
269
270             //liquidate: repay bad debts
271             IERC20(bitmapToken).transferFrom(msg.sender, address (this), badDebts);
272
273             emit LiquidateRent(msg.sender, _rentId, StoppedState.AbnormalLiquidated, 0, badDebts);
274             return;
275         }
```

**Alleviation** Mitigated

The team reply: The liquidation logic has two parts. One part is profitable and anyone can liquidate it. The other part is bad debt, which is liquidated by the team.

## Low risk (4)

### 1. If 1% of the Deposit is Less Than the Gas Fee, None of Liquidator Will Liquidate a Rent

Low risk | Security Analyzer

The `liquidateRent` function allow liquidator to liquidate a rent once its rent fee reaches 99%, so there is atmost 1% of deposit can be the profit for the liquidator.

The concern is that if the deposit is small, so does the profit for its liquidate.

E.g. If the deposit is 1e4 $bitmap, the liquidate profit is 100 $bitmap, if the 100 $bitmap can not cover the gas fee, especially when the blockchain is congested, none of liquidator will liquidate the rent. Meanwhile, the liquidate profit will decrease due to the rent fee increases as time goes by.

Finally, the small deposit will incur more and more bad debt.

**File(s) Affected**

contracts/BitmapRent.sol #253-278

```
    function liquidateRent(string memory _rentId) external whenNotPaused nonReentrant {
        require(!rentIdToRent[_rentId].stopped, "rent already terminated");

        Rent storage rent = rentIdToRent[_rentId];

        //update stat
        _updateStopRentStat(rent.deposit);

        rent.stopped = true;
        rent.stopTimestamp = block.timestamp;
        rent.rentFee = _calRentFee(rent);

        //excessive rent fee
        if (rent.rentFee > rent.deposit) {
            rent.stoppedState = StoppedState.AbnormalLiquidated;
            uint256 badDebts = rent.rentFee-rent.deposit;

            //liquidate: repay bad debts
            IERC20(bitmapToken).transferFrom(msg.sender, address (this), badDebts);

            emit LiquidateRent(msg.sender, _rentId, StoppedState.AbnormalLiquidated, 0, badDebts);
            return;
        }

        uint256 liquidated = rent.deposit - rent.rentFee;
        require(liquidated <= _dailyRentFee(rent.deposit), "It is not time for liquidation");
```

contracts/BitmapRent.sol #580-582

```
    function _dailyRentFee(uint256 rentAmount) internal view returns (uint256) {
        return rentAmount * currentDailyRentFeeRate / FEE_RATE_SCALE_FACTOR;
    }
```

**Alleviation** Mitigated

The team reply that the bad debt will is liquidated by the team.

## 2. The Implementation of `verifyRentSignature` Missing a Few Good Practices

Low risk  Security Analyzer

The `verifyRentSignature` function missing validate the `_expiration` to check if the signature expired or not.

The `verifyRentSignature` function missing using `nounce[msg.sender]` to prevent the potential replay attack.

Refernece: https://eips.ethereum.org/EIPS/eip-2612

**File(s) Affected**

contracts/BitmapRent.sol #335-340

```
    function verifyRentSignature(string memory _rentId, uint256 _firstBitmap, uint256 _n, uint256 _exp
        bytes memory data = abi.encode(msg.sender, _rentId, _firstBitmap, _n, _expiration);
        bytes32 hash = keccak256(data);
        address receivedAddress = ECDSA.recover(hash, _signature);
        return receivedAddress != address(0) && receivedAddress == signer;
    }
```

**Alleviation** Acknowledged

The team acknowledged this finding.

## 3. The `disableInitializers` Function is Missed for Upgradeable Contracts

Low risk | Security Analyzer

The contracts `MerlStake`, and `BitmapRent` are using proxy patterns. The implementation contract behind a proxy can be initialized by any address. This is not a security problem in the sense that it impacts the system directly, as the attacker will not be able to cause any contract to self-destruct or modify any value in the proxy contract. However, taking ownership of implementation contracts can open other attack vectors, like social engineer or phishing attack.

See docs: https://docs.openzeppelin.com/contracts/4.x/api/proxy#Initializable-_disableInitializers--

**File(s) Affected**

contracts/BitmapRent.sol #8-8

```
contract BitmapRent is OwnableUpgradeable {
```

contracts/MerlStake.sol #8-8

```
contract MerlStake is OwnableUpgradeable {
```

**Recommendation**

It is recommended to use this to lock implementation contracts that are designed to be called through proxies.

**Alleviation** Fixed

The team fixed this finding with the commit dd9d38

## 4. The `getTotalReward` Does not Exclude the Bad Debt

Low risk | Security Analyzer

The amount of the total reward return by the `getTotalReward` function does not exclude the bad debt, results in its result being bigger than explected.

More detail can be found in the finding: The `withdrawer` May Withdraw More Rent Fee Than Expected

**File(s) Affected**

contracts/BitmapRent.sol #365-369

```
    function getTotalReward() public view returns (uint256) {
        uint256 interval = block.timestamp - rentStat.updateTimestamp;
        uint256 feePerSecond = rentStat.totalRentDeposit * currentDailyRentFeeRate / FEE_RATE_SCALE_FA
        return rentStat.totalRentFee + feePerSecond * interval;
    }
```

**Recommendation**

Consider excluding the bad debt for the `getTotalReward` function

**Alleviation** Mitigated

The team replied that the last one or two withdrawals may have an impact, and we can top up the contract to mitigate the issue.

## Audit Scope

| File | SHA256 | File Path |
| --- | --- | --- |
| BitmapRent.sol | a4450931f9b3503b14b9aa8ac0d47c08cfd8991399dd6d9d1d2cfb359892a5e2 | /contracts/BitmapRent.sol |
| MerlStake.sol | 584dfc8b8b5f1b469ab8a815af7defa45d10c3518e7afb3602bd64752623d03d | /contracts/MerlStake.sol |

## Disclaimer

This report is governed by the stipulations (including but not limited to service descriptions, confidentiality, disclaimers, and liability limitations) outlined in the Services Agreement, or as detailed in the scope of services and terms provided to you, the Customer or Company, within the context of the Agreement. The Company is permitted to use this report only as allowed under the terms of the Agreement. Without explicit written permission from MetaTrust, this report must not be shared, disclosed, referenced, or depended upon by any third parties, nor should copies be distributed to anyone other than the Company.

It is important to clarify that this report neither endorses nor disapproves any specific project or team. It should not be viewed as a reflection of the economic value or potential of any product or asset developed by teams or projects engaging MetaTrust for security evaluations. This report does not guarantee that the technology assessed is completely free of bugs, nor does it comment on the business practices, models, or legal compliance of the technology's creators.

This report is not intended to serve as investment advice or a tool for investment decisions related to any project. It represents a thorough assessment process aimed at enhancing code quality and mitigating risks inherent in cryptographic tokens and blockchain technology. Blockchain and cryptographic assets inherently carry ongoing risks. MetaTrust's role is to support companies and individuals in their security diligence and to reduce risks associated with the use of emerging and evolving technologies. However, MetaTrust does not guarantee the security or functionality of the technologies it evaluates.

MetaTrust's assessment services are contingent on various dependencies and are continuously evolving. Accessing or using these services, including reports and materials, is at your own risk, on an as-is and as-available basis. Cryptographic tokens are novel technologies with inherent technical risks and uncertainties. The assessment reports may contain inaccuracies, such as false positives or negatives, and unpredictable outcomes. The services may rely on multiple third-party layers.

All services, labels, assessment reports, work products, and other materials, or any results from their use, are provided "as is" and "as available," with all faults and defects, without any warranty. MetaTrust expressly disclaims all warranties, whether express, implied, statutory, or otherwise, including but not limited to warranties of merchantability, fitness for a particular purpose, title, non-infringement, and any warranties arising from course of dealing, usage, or trade practice. MetaTrust does not guarantee that the services, reports, or materials will meet specific requirements, be error-free, or be compatible with other software, systems, or services.

Neither MetaTrust nor its agents make any representations or warranties regarding the accuracy, reliability, or currency of any content provided through the services. MetaTrust is not liable for any content inaccuracies, personal injuries, property damages, or any loss resulting from the use of the services, reports, or materials.

Third-party materials are provided "as is," and any warranty concerning them is strictly between the Customer and the third-party owner or distributor. The services, reports, and materials are intended solely for the Customer and should not be relied upon by others or shared without MetaTrust's consent. No third party or representative thereof shall have any rights or claims against MetaTrust regarding these services, reports, or materials.

The provisions and warranties of MetaTrust in this agreement are exclusively for the Customer's benefit. No third party has any rights or claims against MetaTrust regarding these provisions or warranties. For clarity, the services, including any assessment reports or materials, should not be used as financial, tax, legal, regulatory, or other forms of advice.