BEOSIN
Blockchain Security

Bitmap.Game

Smart Contract Security Audit

No. 202408081835

Aug 8th, 2024

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 1 High-risk, 1 Medium-risk, 4 Low-risk and 1 Info-risk item were identified in the Bitmap.Game project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

- **Risk Description:**

The signature design does not include the chainId, which could pose a security vulnerability of signature replaying if the project is deployed on multiple chains. It is recommended not to use the same signature mechanism on other chains.

- **Project Description:**

The Bitmap.Game project audited in this review primarily comprises three contracts: BitmapRent, MerlStake, and BitmapRentHelperContract. The BitmapRent and BitmapRentHelperContract contracts handle Rent-related functionalities, while the MerlStake contract deals with staking-related functionalities.

The BitmapRent contract contains the main Rent-related functions. Users can call the `startRent` function and transfer the corresponding amount of `bitmapToken` to start the Rent. They can also call the `stopRent` function to end the Rent and redeem the `bitmapToken` after deducting the interest. The Owner can call the `updateRentFeeRate` function to add new fee rates, and subsequent Rent users will be charged in segments based on the effective periods of the historical fee rates. The collected fees will be used as staking rewards for the MerlStake contract.

Additionally, the BitmapRentHelperContract contract is primarily used to query the validity of NFTs. Users can call the `bitmapsRentAvailable` function to check if the specified _bitmaps corresponding to the bitmapNFT can be rented.

Users can call the `MerlStake` function and transfer the merlContract specified tokens to stake. They can then call the `unstakeMerl` function to unstake and receive the rewards generated from multiple rewardContract sources.

1 Overview

1.1 Project Overview

Project Name	Bitmap.Game
Project Language	Solidity
Platform	Merlin
File Hash (SHA-256)	5bf01249bef928788be24ca3ed9722a2db09935880ae5f5e6f40d13e341c36f4(inital)
Code Base	https://github.com/bitmap-game/bitmap-contracts
Commit	9d29a653462213b15d0fbb9e4b55ff2436d466e0 2999271418d4eac5dc9957a230bb955fae24785a 57e92b176822a3eb63b39c05081bc1e96e70c997(final)

1.2 Audit Overview

Audit work duration: Aug 06, 2024 – Aug 08, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's 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.

2. 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.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
Bitmap.Game-01	Signature verification calculation error	High	Fixed
Bitmap.Game-02	Defaults lead to inaccurate rewards	Medium	Fixed
Bitmap.Game-02	Risks in Signature Design	Low	Acknowledged
Bitmap.Game-03	Incorrect calculation of RentReturned	Low	Fixed
Bitmap.Game-04	_getValidRewardContract Calculation Error	Low	Fixed
Bitmap.Game-05	The funds may get locked in the contract	Low	Acknowledged
Bitmap.Game-06	Gas Optimization	Info	Fixed

Finding Details:

[Bitmap.Game-01] Signature verification calculation error

Severity Level	High
Type	Business Security
Lines	BitmapRent.sol#L551-573
Description	The `splitSignature` function can correctly parse the signature's r, s, and v values, and there is no need for further intermediate processing during signature verification, such as the incorrect method of adding 27 to the v value. This incorrect calculation method will cause the originally correct signature verification to fail. At the same time, due to the extensibility attack inherent in the ECDSA signature algorithm, directly using recover may result in two different signatures passing the verification.

```
function recoverSigner(bytes32 hash, bytes memory sig) internal view
returns (bool) {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);
        return ecrecover(hash, v + 27, r, s) == signer;
    }

function splitSignature(bytes memory sig) internal pure returns (uint8,
bytes32, bytes32) {
        require(sig.length == 65);
        bytes32 r;
        bytes32 s;
        uint8 v;
        assembly {
        // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
        // second 32 bytes
            s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes)
            v := byte(0, mload(add(sig, 96)))
        }
        return (v, r, s);
}
```

Recommendation	It is recommended to use the OpenZeppelin ECDSA library to perform signature

	verification.
Status	**Fixed.** The project team used the OpenZeppelin ECDSA library for signature verification.

[Bitmap.Game-02] Defaults lead to inaccurate rewards

Severity Level	**Medium**
Type	Business Security
Lines	BitmapRent.sol#L277-288
Description	Since there is no specified lease expiration time, if the user's rental period is too long, the accumulated fees may exceed the deposit principal. However, the user cannot stop the rental, and the project also lacks the relevant liquidation logic to update the `totalRentDeposit`. This could lead to the continuous accumulation of reward fees exceeding the deposit principal.
Recommendation	It is recommended to add relevant liquidation logic in the project, or ensure that the reward distribution does not exceed the user's deposit principal.
Status	**Fixed.** The project team cliams they will perform timely liquidations to ensure the correct calculation of rewards.

```
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);
}
```

```
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;
        }
...
        emit LiquidateRent(msg.sender, _rentId,
StoppedState.Liquidated, rent.liquidated, 0);
    }
```

[Bitmap.Game-03] Risks in Signature Design

Severity Level	Low
Type	Business Security
Lines	BitmapRent.sol#L265-269

Description	The signature does not include key fields such as the chainId, if the project is deployed on multiple different chains, it may lead to the reuse of signatures on other chains, potentially resulting in replay attacks.
	`function verifyRentSignature(string memory _rentId, uint256 _firstBitmap, uint256 _n, uint256 _expiration, bytes calldata _signature) public view returns (bool){`
	`bytes memory data = abi.encode(msg.sender, _rentId, _firstBitmap, _n, _expiration);`
	`bytes32 hash = keccak256(data);`
	`return recoverSigner(hash, _signature);`
	`}`

Recommendation	It is recommended to refer to the EIP-712 standard when designing the signature specification, in order to properly standardize the signature design.
Status	**Acknowledged.** The project team The project team stated that the signatures are for the backend services, not user signatures. Additionally, the leasing contract is for Merlin users and does not have any cross-chain requirements.

[Bitmap.Game-04] Incorrect calculation of RentReturned

Severity Level	Low
Type	Business Security
Lines	BitmapRent.sol#L235-259
Description	When users call the `getRentReturned` or `getRentsReturned` functions to query the refund amount after deducting the rental fees, there is no verification to check whether the `_rentId` corresponding rent has already ended. Instead, the current timestamp is used to calculate the fee, which may result in the calculated user refund amount being smaller than the actual amount.

```
function getRentReturned(string calldata _rentId) public view
returns(uint256) {
    Rent storage rent = rentIdToRent[_rentId];
    uint256 rentFee = _calRentFee(rent);
    return rent.deposit - rentFee;
}
function getRentsReturned(string[] calldata _rentIds) external view
returns(uint256[] memory) {
    require(_rentIds.length > 0, "invalid _rentIds");
    uint256[] memory returnedList = new uint256[](_rentIds.length);
    for (uint16 i = 0; i < _rentIds.length; i++) {
        returnedList[i] = getRentReturned(_rentIds[i]);
    }
    return returnedList;
}
```

Recommendation	It is recommended to validate whether the `_rentId` has already ended the rent during the query. If the rent has already ended, the rent.returned value should be directly returned.
Status	**Fixed.**

```
function getRentReturned(string calldata _rentId) public view
returns(uint256) {
    Rent memory rent = rentIdToRent[_rentId];
    if (rent.renter == address (0)) {
        return 0;
    }

    if (rent.stopped) {
```

```
        return rent.returned;
    }
    uint256 rentFee = _calRentFee(rent);
    return rent.deposit - rentFee;
}
```

[Bitmap.Game-05] _getValidRewardContract Calculation Error

Severity Level	Low
Type	Business Security
Lines	BitmapRent.sol#L288-304
Description	In the `_getValidRewardContract` function, when calculating the valid reward contracts, the first loop counted the number of valid RewardContracts, but the second loop still used the global iteration index i to assign values, which can lead to an out-of-bounds assignment and cause an exception to be thrown when the number of valid RewardContracts is less than the total number of reward contracts.

```
function _getValidRewardContract() internal view returns (address[]
memory) {
    uint16 count = 0;
    for (uint16 i=0; i<globalRewardContracts.length; i++) {
        if (globalRewards[globalRewardContracts[i]].enabled) {
            count += 1;
        }
    }
    address[] memory contracts = new address[](count);
    for (uint16 i=0; i<globalRewardContracts.length; i++) {
        if (globalRewards[globalRewardContracts[i]].enabled) {
            contracts[i] = globalRewardContracts[i];
        }
    }
    return contracts;
}
```

Recommendation	It is recommended to use two separate indices - one to iterate through all the reward contracts, and another to keep track of the index of the valid reward contracts.
Status	**Fixed.**

```
function _getValidRewardContract() internal view returns (address[]
memory) {
    uint16 count = 0;
    for (uint16 i=0; i<globalRewardContracts.length; i++) {
        if (globalRewards[globalRewardContracts[i]].enabled) {
```

```
            count += 1;
        }
    }
    address[] memory contracts = new address[](count);
    uint16 j = 0;
    for (uint16 i=0; i<globalRewardContracts.length; i++) {
        if (globalRewards[globalRewardContracts[i]].enabled) {
            contracts[j] = globalRewardContracts[i];
            j += 1;
        }
    }
    return contracts;
}
```

[Bitmap.Game-06] The funds may get locked in the contract

Severity Level	Low
Type	Business Security
Lines	BitmapRent.sol#L345-366
Description	Regardless of whether anyone is staking or not, the reward contract will still issue rewards to the staking contract. As long as the total staked amount, `totalMerl` is 0, when a user later stakes, the `_settleGlobalReward` function in the contract will claim the rewards and update `totalRewardsEarned`. However, if `scaledTotalRewardsPerMerl` is 0 at that time, it means this portion of the rewards will not be distributed to the users, and that money will be effectively locked in the contract.

```
function _settleGlobalReward(address rewardContract) internal {
    GlobalReward storage globalReward =
globalRewards[rewardContract];
    uint256 totalReward =
IRewardContract(rewardContract).getTotalReward();
    if (totalReward == 0) {
        globalReward.updateTimestamp = block.timestamp;
        return;
    }

    uint256 rangeReward = totalReward -
globalReward.totalRewardsEarned;
    uint256 scaledRangeRewardPerMerl = 0;
    if (totalMerl > 0) {
        scaledRangeRewardPerMerl =
_scaledRangeRewardPerMerl(rangeReward, totalMerl);
    }
    globalReward.scaledTotalRewardsPerMerl +=
scaledRangeRewardPerMerl;
    globalReward.totalRewardsEarned = totalReward;
    globalReward.updateTimestamp = block.timestamp;
    //withdraw reward from rewardContract.
    if (rangeReward > 0) {
        IRewardContract(rewardContract).withdrawReward(rangeRewar
```

```
d);
    }
}
```

Recommendation	It is recommended that when `totalMerl` is 0, the function should simply update the timestamp and return directly, in order to avoid the issue of funds getting locked in the contract.
Status	**Acknowledged.** The project team has stated that this is based on the business logic, as the rewards are distributed evenly after that.

[Bitmap.Game-07] Gas Optimization

Severity Level	Info
Type	Business Security
Lines	BitmapRent.sol#L235-240

Description	Using the storage keyword in query functions will cause the function to read and modify state variables, which is typically much more expensive than reading and modifying memory variables.
	`function getRentReturned(string calldata _rentId) public view returns(uint256) {`
	`    Rent storage rent = rentIdToRent[_rentId];`
	`    uint256 rentFee = _calRentFee(rent);`
	`    return rent.deposit - rentFee;`
	`}`

Recommendation	It is recommended to replace the `storage` with `memory`.
Status	**Fixed.** The project team replace the `Rent storage rent` with `Rent memory rent` to optimize gas usage.

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.

Likelihood \ Impact	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 generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.3 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.4 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

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com