BLOCKHAT
SECURITY

Gamezland

Smart Contract Security Audit

Prepared by BlockHat

september 22nd, 2022 - september 23rd, 2022

BlockHat.io

contact@blockhat.io

Document Properties

Client	Gurpreet Chauhan
Version	1.0
Classification	Confidential

Scope

The Gamezland Contract in the Gamezland Repository

Files	MD5 Hash
contract/gamezland.sol	5ebf9fcda2f69dbc2e62115a2ec18dbe

Contacts

COMPANY	CONTACT
BlockHat	contact@blockhat.io

Contents

1 Introduction

Gamezland engaged BlockHat to conduct a security assessment on the Gamezland beginning on september 22nd, 2022 and ending september 23rd, 2022. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Gamezland

GAMEZLAND is a GameFi Metaverse with NFT economy powered by $GAME Token. GAMEZLAND citizens will be able to play various p2e games and earn $GAME, own game items as NFTs and trade them, visit NFT exhibitions and 3D NFT marketplaces, buy avatars, weapon skins, virtual estate, clothes, as well as attend virtual concerts, runway shows and other events.

Issuer	Gurpreet Chauhan
Website	`www.gamezland.io`
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

BlockHat used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

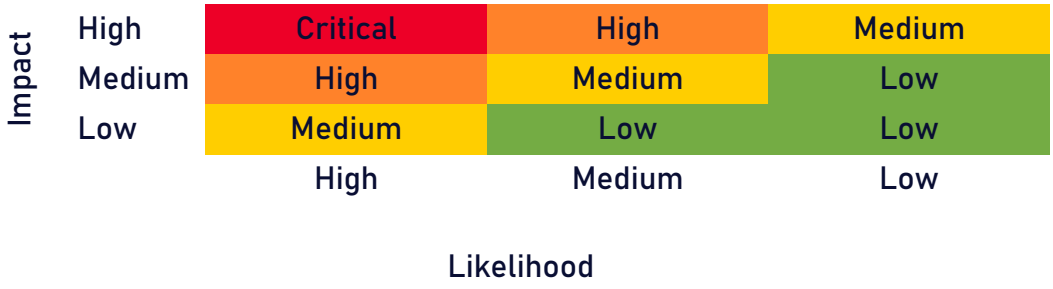
1.2.1 Risk Methodology

Vulnerabilities or bugs identified by BlockHat are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.



2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Gamezland implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 2 critical-severity, 2 high-severity, 1 medium-severity, 2 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
burnFrom() is a public function	CRITICAL	Fixed
Error in function logic	CRITICAL	Fixed
Centralisation risk	HIGH	Fixed
Mint should be locked for 2 years	HIGH	Fixed
Usage of tx.origin	MEDIUM	Fixed
Floating Pragma	LOW	Fixed
Missing address verification	LOW	Fixed

3 Finding Details

A gamezland.sol

A.1 burnFrom() is a public function [CRITICAL]

Description:

Anyone can call burnFrom function and burn tokens from any address he want. This represents a big risk on the user and the owner side.

Code:

Listing 1: gamezland.sol

```
82      function burnFrom(address from, uint amount) public {
83          require(amount <= balances[from], 'More than the balance!');
84          require(amount <= allowed[from][msg.sender], 'More than allowed
                ↪ !');

86          totalSupply -= amount;
87          balances[from] -= amount;
88          allowed[from][msg.sender] -= amount;

90          emit Transfer(from, address(0), amount);
91      }
```

Risk Level:

Likelihood – 5
Impact - 5

Recommendation:

We suggest to restrict this function to only the owner.

Status - Fixed

The dev Team fixed the issue

A.2 Error in function logic [CRITICAL]

Description:

The require statement is always false unless if amount=0 , so mint() will never function.

Code:

Listing 2: gamezland.sol

```
63      function mint(address recipient, uint amount) public {
64          require(msg.sender == minter, 'Only minter can do this!');
65          require(totalSupply + amount >= totalSupply);
67          totalSupply += amount;
68          balances[recipient] += amount;
70          emit Transfer(address(0), recipient, amount);
71      }
```

Risk Level:

Likelihood – 5
Impact - 5

Recommendation:

We recommend removing this require statement. If it is meant that the amount should not exceed a max supply, you should add a maxsupply variable and define it in the contructor.

Status - Fixed

The dev Team fixed the issue

A.3 Centralisation risk [HIGH]

Description:

Besides that this fuction is public. If the owner restrict the function to his self he will have super control over the tokens distrubution This represents a centralisation risk.

Code:

Listing 3: gamezland.sol

```
82      function burnFrom(address from, uint amount) public {
83          require(amount <= balances[from], 'More than the balance!');
84          require(amount <= allowed[from][msg.sender], 'More than allowed
                ↪ !');

86          totalSupply -= amount;
87          balances[from] -= amount;
88          allowed[from][msg.sender] -= amount;

90          emit Transfer(from, address(0), amount);
91      }
```

Risk Level:

Likelihood – 5
Impact - 4

Recommendation:

We suggest removing this function.

Status – Fixed

The dev Team fixed the issue

A.4 Mint should be locked for 2 years [HIGH]

Description:

As the white paper states mint() function should be restricted for 2 years. If not this will remove trust between the owner and the users.

Code:

Listing 4: gamezland.sol

```
82      function burnFrom(address from, uint amount) public {
83          require(amount <= balances[from], 'More than the balance!');
84          require(amount <= allowed[from][msg.sender], 'More than allowed
                ↪ !');

86          totalSupply -= amount;
87          balances[from] -= amount;
88          allowed[from][msg.sender] -= amount;

90          emit Transfer(from, address(0), amount);
91      }
```

Risk Level:

Likelihood – 4
Impact - 3

Recommendation:

We suggest to add a require statement that you can only use this function after 2 years of deploying the smart contract.

Status - Fixed

The dev Team fixed the issue

A.5 Usage of tx.origin [MEDIUM]

Description:

Never use tx.origin for authorization, another contract can have a method which will call your contract (where the user has some funds for instance) and your contract will authorize that transaction as your address is in tx.origin.

Code:

Listing 5: gamezland.sol

```
20      constructor(string memory _name, string memory _symbol, uint _supply
            ↪ , uint _dec, address _owner) {
21          name = _name;
22          symbol = _symbol;
23          decimals = _dec;
24          totalSupply = _supply * 10 ** _dec;
25          balances[_owner] = totalSupply;
26          minter = tx.origin;
27          emit Transfer(address(0), _owner, totalSupply);
28      }
```

Risk Level:

Likelihood – 3
Impact - 3

Recommendation:

You should use msg.sender for authorization (if another contract calls your contract msg.sender will be the address of the contract and not the address of the user who called the contract).

Status - Fixed

The dev Team fixed the issue

A.6 Floating Pragma [LOW]

Description:

The contract makes use of the floating-point pragma 0.8.2 . Contracts should be deployed using the same compiler version and flags that were used during the testing process.Locking the pragma helps ensure that contracts are not unintentionally deployed using another pragma, such as an obsolete version, that may introduce issues in the contract system.

Code:

Listing 6: gamezland.sol

```
5 pragma solidity ^0.8.2;
```

Risk Level:

Likelihood – 1
Impact - 3

Recommendation:

Consider locking the pragma version. It is advised that floating pragma should not be used in production. Both truffle-config.js and hardhat.config.js support locking the pragma version.

Status - Fixed

The dev Team fixed the issue.

A.7 Missing address verification [LOW]

Description:

Certain functions lack a safety check in the address, the address-type argument _owner should include a zero-address test, otherwise, the contract's functionality may become inaccessible.

Code:

Listing 7: gamezland.sol

```
20  constructor(string memory _name, string memory _symbol, uint _supply,
        ↪ uint _dec, address _owner) {
```

Risk Level:

Likelihood – 1
Impact - 3

Recommendation:

We recommend that you make sure the addresses provided in the arguments are different from the address(0).

Status - Fixed

The dev Team fixed the issue.

4 Best Practices

BP.1 SPDX license identifier not provided in source file.

Description:

SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code.

BP.2 Arguments initialization

Description:

It is better to initialize Arguments to prevent errors.

Code:

Listing 8: gamezland.sol

```
20      constructor(string memory _name, string memory _symbol, uint _supply
            ↪ , uint _dec, address _owner) {
21          name = _name;
22          symbol = _symbol;
23          decimals = _dec;
24          totalSupply = _supply * 10 ** _dec;
25          balances[_owner] = totalSupply;
26          minter = tx.origin;
27          emit Transfer(address(0), _owner, totalSupply);
28      }
```

5 Static Analysis (Slither)

Description:

Block Hat expanded the coverage of the specific contract areas using automated testing methodologies. Slither, a Solidity static analysis framework, was one of the tools used. Slither was run on all-scoped contracts in both text and binary formats. This tool can be used to test mathematical relationships between Solidity instances statically and variables that allow for the detection of errors or inconsistent usage of the contracts' APIs throughout the entire codebase.

Results:

```
Compilation warnings/errors on gamezland.sol:
Warning: SPDX license identifier not provided in source file. Before
    ↪ publishing, consider adding a comment containing "SPDX-License-
    ↪ Identifier: <SPDX-License>" to each source file. Use "SPDX-
    ↪ License-Identifier: UNLICENSED" for non-open-source code. Please
    ↪ see https://spdx.org for more information.
--> gamezland.sol



Pragma version^0.8.2 (gamezland.sol#5) allows old versions
solc-0.8.12 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #incorrect-versions-of-solidity

balanceOf(address) should be declared external:
        - Token.balanceOf(address) (gamezland.sol#30-32)
transfer(address,uint256) should be declared external:
        - Token.transfer(address,uint256) (gamezland.sol#34-40)
transferFrom(address,address,uint256) should be declared external:
```

```
        - Token.transferFrom(address,address,uint256) (gamezland.sol
          ↪ #42-50)
approve(address,uint256) should be declared external:
        - Token.approve(address,uint256) (gamezland.sol#52-56)
allowance(address,address) should be declared external:
        - Token.allowance(address,address) (gamezland.sol#58-60)
mint(address,uint256) should be declared external:
        - Token.mint(address,uint256) (gamezland.sol#63-71)
burn(uint256) should be declared external:
        - Token.burn(uint256) (gamezland.sol#73-80)
burnFrom(address,uint256) should be declared external:
        - Token.burnFrom(address,uint256) (gamezland.sol#82-91)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
   ↪ #public-function-that-could-be-declared-external
gamezland.sol analyzed (1 contracts with 78 detectors), 10 result(s)
   ↪ found
```

Conclusion:

Most of the vulnerabilities found by the analysis have already been addressed by the smart contract code review.

6 Conclusion

We examined the design and implementation of Gamezland in this audit and found several issues of various severities. We advise Gurpreet Chauhan team to implement the recommendations contained in all 7 of our findings to further enhance the code's security. It is of utmost priority to start by addressing the most severe exploit discovered by the auditors then followed by the remaining exploits, and finally we will be conducting a re-audit following the implementation of the remediation plan contained in this report.

We would much appreciate any constructive feedback or suggestions regarding our methodology, audit findings, or potential scope gaps in this report.

BLOCKHAT
SECURITY

For a Smart Contract Audit, contact us at contact@blockhat.io