BLOCKSEC

Security Audit

Report for Cellula Life Game V2 and CELL Token

Date: July 25, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Cellula
Target	Cellula Life Game V2 and CELL Token

Version History

Version	Date	Description
1.0	June 25, 2024	First release

Signature

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on the Cellula Life Game V2[1] and `CELL` token[2] of the Cellula. The Cellula introduces a Full-Chain Game that combines two types of NFTs: `BitCell` and `BitLife`. The issuance curve is determined by the `VRGDA` method, with `BitCell` having a fixed total supply of 10,220 units and `BitLife` having no upper limit on the total supply. The game rules are based on the logic of Conway's Game of Life, where each `BitCell` is a 3x3 matrix, and each `BitLife` NFT requires a combination of 2 to 9 `BitCells` to be minted.

Please note that the audit scope is limited to the following smart contracts:

```
contract-energy-factory-v2/contracts/V2/CellGameV2.sol
contract-energy-factory-v2/contracts/V2/LifeV2.sol
contract-cell-token/contracts/CellulaToken.sol
```

Listing 1.1: Audit Scope for this Report

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Cellula Life Game V2	`Version 1`	`cb65597e3c22860c27a6807d69f0df17d2326945`
	`Version 2`	`cc3b11d82516535c56b38d4a1b2536b995315d69`
CELL Token	`Version 1`	`c182ee24658f7505953a693b1830810b0915c737`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset.

[1] https://github.com/cellulalifegame/contract-energy-factory-v2

[2] https://github.com/cellulalifegame/contract-cell-token

Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic

- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **three** recommendations and **one** note.

- Medium Risk: 2
- Low Risk: 1
- Recommendation: 3
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Incorrect index update of `token_ids`	Software Security	Fixed
2	Low	Lack of refund in function `buyFood()`	DeFi Security	Fixed
3	Medium	Lack of `payToken` check in function `setLifeCreationConfig()`	DeFi Security	Fixed
4	-	Lack of balance check in function `setPayTokenAddress()`	Recommendation	Fixed
5	-	Lack of following Checks Effects Interactions pattern	Recommendation	Fixed
6	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
7	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect index update of `token_ids`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `batchCreateLife()`, the update index for `token_ids` incorrectly runs from 0 to the gene's quantity number. This is incorrect and a new round of token IDs will override previous ones.

```
783   function batchCreateLife(uint256[][][] calldata genes, uint256[] calldata quantitys) public
          payable returns(uint256[] memory token_ids) {
784       require(pauseStatus == false, "pause");
785       require(genes.length == quantitys.length, 'params length error');
786
787       // token_ids
788       uint256 quantityTotal;
789       for (uint256 i = 0; i < quantitys.length; ++i) {
790           quantityTotal += quantitys[i];
791       }
792       token_ids = new uint256[](quantityTotal);
```

```
793
794
795         uint256 cumulatedPrice = 0; // Total value of all life
796         uint256 totalRentFeeCollected = 0;
797
798
799         // handler each gene
800         for (uint256 i = 0; i < genes.length; ++i) {
801             uint256 quantity = quantitys[i];
802             uint256[][] calldata cellsPositions_ = genes[i];
803             require( cellsPositions_.length >= 2 && cellsPositions_.length <= 9, "can only use 2-9
                    cells!");
804
805
806             // data for cellGenes and livingCellTotals
807             uint256[] memory cellGenes = new uint256[](cellsPositions_.length);
808             uint32[] memory livingCellTotals = new uint32[](cellsPositions_.length);
809             for (uint256 j = 0; j < cellsPositions_.length; ++j) {
810                 CellGene storage cellGene = _cellPool[cellsPositions_[j][0]];
811                 require(cellGene.bornTime > 0, "cell not minted");
812                 cellGenes[j] = cellGene.bitmap.getBucket(0);
813                 livingCellTotals[j] = cellGene.livingCellTotal;
814             }
815
816
817             // mint ‘quantity‘ life using single gene
818             for (uint256 index = 0; index < quantity; ++index) {
819                 uint256 lifeBornPrice;
820
821
822                 // got cells price
823                 for (uint256 j = 0; j < cellsPositions_.length; ++j) {
824                     uint256 tokenId = cellsPositions_[j][0];
825                     CellGene storage cellGene = _cellPool[tokenId];
826                     uint256 absoluteTimeSinceStart = block.timestamp - cellGene.bornTime;
827                     uint256 cellRentPrice = getCellRentPrice( cellGene.rentedCount,
                            absoluteTimeSinceStart);
828                     cellGene.rentedCount += 1;
829
830
831                     lifeBornPrice += cellRentPrice;
832                     cumulatedPrice += cellRentPrice;
833                     uint256 rentFee = (cellRentPrice * 5) / 100;
834                     if (tokenPaymentStatus) {
835                         _rentTokenFeeCollected[tokenId] += rentFee;
836                     } else {
837                         _rentFeeCollected[tokenId] += rentFee;
838                     }
839                     totalRentFeeCollected += rentFee;
840                     emit MintFeeReceived(tokenId, rentFee);
841                 }
842
843                 token_ids[index] = _life.createLife(msg.sender, lifeBornPrice, cellsPositions_,
```

```
                    cellGenes, livingCellTotals);
844             lifeBornPrice = 0;
845         }
846     }
```

Listing 2.1: contracts/V2/CellGameV2.sol

Impact This error results in incorrect updating of `token_ids` , potentially causing data inconsistency and the loss of reference to earlier tokens.

Suggestion The index for updating `token_ids` should correctly run from 0 to `quantityTotal`, ensuring that each new token ID is unique and does not overwrite existing ones.

2.2 DeFi Security

2.2.1 Lack of refund in function `buyFood()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `buyFood()` lacks the implementation to refund excess native tokens to the user.

```
139   function buyFood(
140       uint256[] memory tokenIds,
141       uint256 foodWorkTime
142   ) external payable {
143       require(pauseStatus == false, "pause");
144       uint256 foodPrice = _foodPrices[foodWorkTime];
145       if (foodPrice <= 0) {
146           revert FoodNotOnSale(foodWorkTime);
147       }
148       uint256 foodPriceSum = tokenIds.length * foodPrice;
149
150
151       // Payment
152       if (tokenPaymentStatus) {
153           require(msg.value == 0, "value must be 0");
154           IERC20(_foodPayToken).safeTransferFrom(address(msg.sender), address(this), foodPriceSum
                  );
155       } else {
156           if (msg.value < foodPriceSum) {
157               revert EtherNotEnough(foodPriceSum);
158           }
159       }
160
161
162       // workEndTime
163       uint256 currentTime = block.timestamp;
164       for (uint256 i = 0; i < tokenIds.length; ++i) {
165           if (publicBuyStatus == false) {
```

```
166             address owner = _ownerOf(tokenIds[i]);
167             if (msg.sender != owner) {
168                 revert MustBeNftOwner(owner);
169             }
170         }
171
172
173         uint256 actualWorkingTime = foodWorkTime;
174         if (_lifePool[tokenIds[i]].workEndTime > currentTime) {
175             actualWorkingTime += (_lifePool[tokenIds[i]].workEndTime - currentTime);
176         }
177         require(actualWorkingTime <= maxWorkTime, "time limit exceeded");
178         _lifePool[tokenIds[i]].workEndTime = uint64(currentTime + actualWorkingTime);
179         emit FeedEvent(tokenIds[i], currentTime, foodWorkTime);
180     }
181 }
```

Listing 2.2: contracts/v2/LifeV2.sol

Impact Users will lose the overpaid native tokens.

Suggestion Add logic in the function `buyFood()` to refund native tokens.

2.2.2 Lack of `payToken` check in function `setLifeCreationConfig()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `setLifeCreationConfig()` can be invoked multiple times in the current contract to set the config. However, the `_totalRentTokenFee` for different `payToken` is recorded in a single state variable. After the `payToken` is updated, users will be unable to withdraw their rent fee or withdraw the wrong tokens.

```
102 function setLifeCreationConfig(
103     int256 soldBySwitch_,
104     int256 switchTime_,
105     int256 cellTargetRentPrice_,
106     int256 priceDecayPercent_,
107     int256 logisticLimit_,
108     int256 timeScale_,
109     int256 perTimeUnit_,
110     uint256 updateInterval_,
111     address payToken_
112 ) public onlyOwner {
113     int256 decayConstant = wadLn(1e18 - priceDecayPercent_);
114     require(decayConstant < 0, "NON_NEGATIVE_DECAY_CONSTANT");
115
116
117     _lifeCreationConfig.soldBySwitch = soldBySwitch_;
118     _lifeCreationConfig.switchTime = switchTime_;
119     _lifeCreationConfig.cellTargetRentPrice = cellTargetRentPrice_;
120     _lifeCreationConfig.decayConstant = decayConstant;
```

```
121         _lifeCreationConfig.logisticLimit = logisticLimit_;
122         _lifeCreationConfig.timeScale = timeScale_;
123         _lifeCreationConfig.perTimeUnit = perTimeUnit_;
124         _lifeCreationConfig.updateInterval = updateInterval_;
125         _lifeCreationConfig.payToken = payToken_;
126     }
```

Listing 2.3: contracts/V2/CellGameV2.sol

Impact Users will be unable to withdraw their rent fee or withdraw the wrong tokens.

Suggestion Ensure the `payToken` can be updated only once.

2.3 Additional Recommendation

2.3.1 Lack of balance check in function `setPayTokenAddress()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `setPayTokenAddress()` doesn't ensure the pay token's balance is zero before it updates the token address. If there are old pay tokens left in the contract, the owner has to reset the pay token address to withdraw the tokens.

```
409     function setPayTokenAddress(address food_pay_token) public onlyOwner {
410         emit ChangePayTokenAddress(_foodPayToken, food_pay_token);
411         _foodPayToken = food_pay_token;
412     }
```

Listing 2.4: contracts/V2/LifeV2.sol

Suggestion Ensure the token balance is zero in the function `setPayTokenAddress()`.

2.3.2 Lack of following Checks Effects Interactions pattern

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `batchCreateLife()`, the external invocation `payable(msg.sender).call` occurs before the state variables `_totalRentTokenFee/_devTokenFeeCollected` are updated, which breaks the (Checks Effects Interactions) `C-E-I` pattern.

The same issue also exists in the function `createLife()`.

```
783     function batchCreateLife(uint256[][][] calldata genes, uint256[] calldata quantitys) public
            payable returns(uint256[] memory token_ids) {
784         require(pauseStatus == false, "pause");
785         require(genes.length == quantitys.length, 'params length error');
786
787         // token_ids
788         uint256 quantityTotal;
789         for (uint256 i = 0; i < quantitys.length; ++i) {
790             quantityTotal += quantitys[i];
```

```
791         }
792         token_ids = new uint256[](quantityTotal);
793 
794 
795         uint256 cumulatedPrice = 0; // Total value of all life
796         uint256 totalRentFeeCollected = 0;
797 
798 
799         // handler each gene
800         for (uint256 i = 0; i < genes.length; ++i) {
801             uint256 quantity = quantitys[i];
802             uint256[][] calldata cellsPositions_ = genes[i];
803             require( cellsPositions_.length >= 2 && cellsPositions_.length <= 9, "can only use 2-9
                    cells!");
804 
805 
806             // data for cellGenes and livingCellTotals
807             uint256[] memory cellGenes = new uint256[](cellsPositions_.length);
808             uint32[] memory livingCellTotals = new uint32[](cellsPositions_.length);
809             for (uint256 j = 0; j < cellsPositions_.length; ++j) {
810                 CellGene storage cellGene = _cellPool[cellsPositions_[j][0]];
811                 require(cellGene.bornTime > 0, "cell not minted");
812                 cellGenes[j] = cellGene.bitmap.getBucket(0);
813                 livingCellTotals[j] = cellGene.livingCellTotal;
814             }
815 
816 
817             // mint ‘quantity‘ life using single gene
818             for (uint256 index = 0; index < quantity; ++index) {
819                 uint256 lifeBornPrice;
820 
821 
822                 // got cells price
823                 for (uint256 j = 0; j < cellsPositions_.length; ++j) {
824                     uint256 tokenId = cellsPositions_[j][0];
825                     CellGene storage cellGene = _cellPool[tokenId];
826                     uint256 absoluteTimeSinceStart = block.timestamp - cellGene.bornTime;
827                     uint256 cellRentPrice = getCellRentPrice( cellGene.rentedCount,
                            absoluteTimeSinceStart);
828                     cellGene.rentedCount += 1;
829 
830 
831                     lifeBornPrice += cellRentPrice;
832                     cumulatedPrice += cellRentPrice;
833                     uint256 rentFee = (cellRentPrice * 5) / 100;
834                     if (tokenPaymentStatus) {
835                         _rentTokenFeeCollected[tokenId] += rentFee;
836                     } else {
837                         _rentFeeCollected[tokenId] += rentFee;
838                     }
839                     totalRentFeeCollected += rentFee;
840                     emit MintFeeReceived(tokenId, rentFee);
841                 }
```

```
842
843             token_ids[index] = _life.createLife(msg.sender, lifeBornPrice, cellsPositions_,
                    cellGenes, livingCellTotals);
844             lifeBornPrice = 0;
845         }
846     }
847
848
849     // collect user funds
850     if (tokenPaymentStatus) {
851         require(msg.value == 0, "value must be 0");
852         IERC20(_lifeCreationConfig.payToken).safeTransferFrom(address(msg.sender), address(this
                ), cumulatedPrice);
853     } else {
854         require(msg.value >= cumulatedPrice, "Insufficient funds");
855         if (msg.value > cumulatedPrice) {
856             (bool sent, ) = payable(msg.sender).call{ value: msg.value - cumulatedPrice }("");
                    // Returns false on failure
857             require(sent, "eth return failed");
858         }
859     }
860
861
862     // for devFee and poolFee
863     uint256 remainFee = cumulatedPrice - totalRentFeeCollected;
864     if (tokenPaymentStatus) {
865         _totalRentTokenFee += totalRentFeeCollected;
866         _devTokenFeeCollected += (remainFee * 14) / 19 ;
867         _poolTokenFeeCollected += remainFee - (remainFee * 14) / 19;
868     } else {
869         _totalRentFee += totalRentFeeCollected;
870         _devFeeCollected += (remainFee * 14) / 19 ;
871         _poolFeeCollected += remainFee - (remainFee * 14) / 19;
872     }
873     emit MintFeeForDevReceived((remainFee * 14) / 19);
874     emit MintFeeForPoolReceived(remainFee - (remainFee * 14) / 19);
875
876
877     // END
878 }
```

Listing 2.5: contracts/v2/CellGameV2.sol

```
186 function createLife(uint256[][] memory cellsPositions_) public payable returns(uint256
        life_token_id) {
187     require(pauseStatus == false, "pause");
188     require(
189         cellsPositions_.length >= 2 && cellsPositions_.length <= 9,
190         "can only use 2-9 cells!"
191     );
192     uint256 cumulatedPrice = 0;
193     uint256[] memory cellGenes = new uint256[](cellsPositions_.length);
194     uint32[] memory livingCellTotals = new uint32[](cellsPositions_.length);
```

```
195
196
197         uint256 totalRentFeeCollected = 0;
198         for (uint256 i = 0; i < cellsPositions_.length; ++i) {
199             uint256 tokenId = cellsPositions_[i][0];
200             CellGene storage cellGene = _cellPool[tokenId];
201             require(cellGene.bornTime > 0, "cell not minted");
202             uint256 absoluteTimeSinceStart = block.timestamp -
203                 cellGene.bornTime;
204             uint256 cellRentPrice = getCellRentPrice(
205                 cellGene.rentedCount,
206                 absoluteTimeSinceStart
207             );
208
209
210             cellGenes[i] = cellGene.bitmap.getBucket(0);
211             livingCellTotals[i] = cellGene.livingCellTotal;
212             cellGene.rentedCount += 1;
213
214
215             cumulatedPrice += cellRentPrice;
216
217
218             uint256 rentFee = (cellRentPrice * 5) / 100;
219
220
221             if (tokenPaymentStatus) {
222                 _rentTokenFeeCollected[tokenId] += rentFee;
223             } else {
224                 _rentFeeCollected[tokenId] += rentFee;
225             }
226             totalRentFeeCollected += rentFee;
227             emit MintFeeReceived(tokenId, rentFee);
228
229
230         }
231
232
233         uint256 remainFee = cumulatedPrice - totalRentFeeCollected;
234
235
236         if (tokenPaymentStatus) {
237             _totalRentTokenFee += totalRentFeeCollected;
238             _devTokenFeeCollected += (remainFee * 14) / 19 ;
239             _poolTokenFeeCollected += remainFee - (remainFee * 14) / 19;
240         } else {
241             _totalRentFee += totalRentFeeCollected;
242             _devFeeCollected += (remainFee * 14) / 19 ;
243             _poolFeeCollected += remainFee - (remainFee * 14) / 19;
244         }
245
246
247         emit MintFeeForDevReceived((remainFee * 14) / 19);
```

```
248         emit MintFeeForPoolReceived(remainFee - (remainFee * 14) / 19);
249
250
251         // Payment
252         if (tokenPaymentStatus) {
253             require(msg.value == 0, "value must be 0");
254             IERC20(_lifeCreationConfig.payToken).safeTransferFrom(address(msg.sender), address(this
                    ), cumulatedPrice);
255         } else {
256             require(msg.value >= cumulatedPrice, "Insufficient funds");
257             if (msg.value > cumulatedPrice) {
258                 (bool sent, ) = payable(msg.sender).call{ value: msg.value - cumulatedPrice }("");
                        // Returns false on failure
259                 require(sent, "eth return failed");
260             }
261         }
262
263
264         life_token_id = _life.createLife(
265             msg.sender,
266             cumulatedPrice,
267             cellsPositions_,
268             cellGenes,
269             livingCellTotals
270         );
271     }
```

Listing 2.6: contracts/v2/CellGameV2.sol

Suggestion Follow the Checks Effects Interactions pattern.

2.3.3 Lack of invoking function `_disableInitializers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Currently, there is no invocation of function `_disableInitializers()` in both the `LifeV2` and `CellGameV2` contracts, which means the implementation contracts can be initialized and can result in potential risks.

Suggestion Invoke the function `_disableInitializers()` in `LifeV2` and `CellGameV2` contracts' constructors.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description In the Cellula Life Game V2 and CELL Token contracts, there are several functions (e.g. `mintCell()`, `setPauseStatus()`, `setCellAddress()`) with privileges to set key pa-

rameters. Modifying these parameters can significantly alter the functionality of the contracts or cause the loss of users.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS