BLOCKSEC

Security Audit

Report for Cellula Life Game Contracts

Date: April 8, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Cellula
Target	Cellula Life Game Contracts

Version History

Version	Date	Description
1.0	April 8, 2024	First Release

Signature

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

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 Contracts of the Cellula [1]. 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:

- src/CellGame.sol
- src/Energy.sol
- src/Helps.sol
- src/Life.sol
- src/interface/ICellGame.sol
- src/interface/ILife.sol
- src/lib/BitMap.sol
- src/lib/SignedWadMath.sol
- src/lib/VRGDA.sol

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 Contracts	`Version 1`	`9be5051fe3471eb44fd1911d17c26d7b5be1a208`
	`Version 2`	`2fc1536504e46b0a14b96664ebbfecde2a2db405`

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/Energy-Factory-Solidity/tree/main

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 [2] and Common Weakness Enumeration [3]. 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

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

[3] 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 find **eight** potential security issues. Besides, we also have **four** recommendation and **four** notes.

- High Risk: 2
- Medium Risk: 6
- Low Risk: 0
- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Potential Gene Manipulation Due to Predictable Randomness	Software Security	Confirmed
2	High	Lack of `tokenId` Check in Function `createLife()`	DeFi Security	Fixed
3	High	Lack of `_currentCellAuction.sold` Update in Function `addNewAuction`	DeFi Security	Fixed
4	Medium	Incorrect Update of `workEndTime`	DeFi Security	Confirmed
5	Medium	Lack of Refund in Function `buyFood()`	DeFi Security	Confirmed
6	Medium	Lack of Check in Function `addNewAuction()`	DeFi Security	Fixed
7	Medium	Lack of Interface to Withdraw `_poolFeeCollected` Fee	DeFi Security	Fixed
8	Medium	Lack of Upper Limit in Function `Withdraw()`	DeFi Security	Fixed
9	-	Lack of Check in Function `createLife()`	Recommendation	Fixed
10	-	Incorrect Comments	Recommendation	Fixed
11	-	Redundant code	Recommendation	Fixed
12	-	Improper usage of function Transfer	Recommendation	Fixed
13	-	Higher Cell Price Due to Round Down Design	Note	-
14	-	Inconsistent `BLOCK_TIME`	Note	-
15	-	Lack of Access Control in `sendClaimEnergyRequest()`	Note	-
16	-	Lack of Evolution Implementation	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential Gene Manipulation Due to Predictable Randomness

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `MintFromAuction()` of the contract `CellGame`, it uses the function `getRandomNumber()` to generate the bitmap (gene) of the cell. However, the randomness is predictable and user-controlled because `msg.sender` is under the user's control, allowing them to calculate an address that yields a specific random number (cell gene).

```
388   function getRandomNumber() public returns (uint256) {
389       uint256 randomNumber = uint256(
390           keccak256(abi.encodePacked(block.timestamp, msg.sender))
391       );
392       for (uint256 i = 0; i < MAX_RANDOM_NUM; i++) {
393           uint256 index = (randomNumber + i) % MAX_RANDOM_NUM;
394           if (!_randomBitmap.get(index)) {
395               _randomBitmap.set(index);
396               return index + 1;
397           }
398       }
399
400
401       _randomBitmap.unsetBucket(0, 0);
402       _randomBitmap.unsetBucket(1, 0);
403       _current_round_number += 1;
404       return getRandomNumber();
405   }
```

Listing 2.1: src/CellGame.sol

Impact Users can mint a specific cell/Life, leading to unfair issues.

Suggestion Use an oracle (e.g., chainlink) to get a random number for the cell gene.

Feedback from the Project This is by design.

2.2 DeFi Security

2.2.1 Lack of `tokenId` Check in Function `createLife()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `createLife()`, it uses the `tokenId` provided by the caller to get the cells to be rented. However, it doesn't make sure the `tokenId` (`cellsPositions_[i][0]`) represents a minted cell, and malicious users can set `tokenId` to a non-minted cell and construct life from it. Because `cellGene.bornTime` is zero, `absoluteTimeSinceStart` will be a huge value, and the rent fee would be rather low. Furthermore, since it will update the `rentedCount` for the (non-minted) cell, when the cell is minted in the future, the user will suffer from indirect loss since the cell's rent fee will be rather high.

```
163   function createLife(uint256[][] memory cellsPositions_) public payable {
164       require(
165           cellsPositions_.length >= 2 && cellsPositions_.length <= 9,
166           "can only use 2-9 cells!"
```

```
167         );
168         uint256 cumulatedPrice = 0;
169         uint256[] memory cellGenes = new uint256[](cellsPositions_.length);
170         uint32[] memory livingCellTotals = new uint32[](cellsPositions_.length);
171
172
173         uint256 totalRentFeeCollected = 0;
174         for (uint256 i = 0; i < cellsPositions_.length; i++) {
175             uint256 tokenId = cellsPositions_[i][0];
176             CellGene storage cellGene = _cellPool[tokenId]; // <== non-exist ID
177             uint256 absoluteTimeSinceStart = block.timestamp -
178                 cellGene.bornTime;
179             uint256 cellRentPrice = getCellRentPrice( // <== low fee
180                 cellGene.rentedCount,
181                 absoluteTimeSinceStart
182             );
183
184
185             cellGenes[i] = cellGene.bitmap.getBucket(0);
186             livingCellTotals[i] = cellGene.livingCellTotal;
187             cellGene.rentedCount += 1; // <== corrupt
188
189
190                 uint256 rentFee = (cellRentPrice * 70) / 100;
191                 _rentFeeCollected[tokenId] += rentFee;
192                 totalRentFeeCollected += rentFee;
193
194
195                 emit MintFeeReceived(tokenId, rentFee);
196
197
198                 cumulatedPrice += cellRentPrice;
199             }
200
201
202             uint256 remainFee = cumulatedPrice - totalRentFeeCollected;
203             _devFeeCollected += remainFee / 3;
204             _poolFeeCollected += remainFee - remainFee / 3;
205
206
207             emit MintFeeForDevReceived(remainFee / 3);
208             emit MintFeeForPoolReceived(remainFee - remainFee / 3);
209
210
211             require(msg.value >= cumulatedPrice, "Insufficient funds");
212
213
214             _life.createLife(
215                 msg.sender,
216                 cumulatedPrice,
217                 cellsPositions_,
218                 cellGenes,
219                 livingCellTotals
```

```
220         );
221
222
223         if (msg.value > cumulatedPrice) {
224             (bool sent, ) = payable(msg.sender).call{
225                 value: msg.value - cumulatedPrice
226             }(""); // Returns false on failure
227             require(sent, "failed to return Ether");
228         }
229     }
```

Listing 2.2: src/CellGame.sol

Impact Users can corrupt any non-minted tokens and create life with low fees.

Suggestion Check if the cell exists in the function `createLife()`.

2.2.2 Lack of `_currentCellAuction.sold` Update in Function `addNewAuction`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `addNewAuction()`, it ensured that the sold cell number was equal to maxSellable, and then updated the fields in `_currentCellAuction`. However, it didn't reset the `_currentCellAuction.sold` to zero.

```
244   function addNewAuction(
245       int256 targetPrice_,
246       int256 priceDecayPercent_,
247       int256 perTimeUnit_,
248       uint256 startTime_,
249       uint256 maxSellable_,
250       uint256 startTokenID_,
251       uint256 updateInterval_
252   ) public onlyOwner {
253       require(
254           _currentCellAuction.sold == _currentCellAuction.maxSellable,
255           "auction ongoing"
256       );
257       require(startTime_ > block.timestamp, "invalid startTime");
258       int256 decayConstant = wadLn(1e18 - priceDecayPercent_);
259       require(decayConstant < 0, "NON_NEGATIVE_DECAY_CONSTANT");
260       _currentCellAuction.startTime = startTime_;
261       _currentCellAuction.targetPrice = targetPrice_;
262       _currentCellAuction.decayConstant = decayConstant;
263       _currentCellAuction.perTimeUnit = perTimeUnit_;
264       _currentCellAuction.maxSellable = maxSellable_;
265       _currentCellAuction.startTokenID = startTokenID_;
266       _currentCellAuction.updateInterval = updateInterval_;
267   }
```

Listing 2.3: src/CellGame.sol

Impact First, users would be charged with a higher rent fee calculated using `getVRGDAPrice()`. Second, users won't be able to mint cells in auction rounds except the first one.

Suggestion Reset `_currentCellAuction.sold` to zero in function `addNewAuction()`.

2.2.3 Incorrect Update of `workEndTime`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description Currently the function `buyFood()` is implemented according to the documentation – "When food is consumed, the work time in BitLife is reset, rather than accumulated." Specifically, it will always set `_lifePool[tokenIds[i]].workEndTime` to `currentTime + foodWorkTime`. However, when a user buys multiple food items at once in the function `buyFood()`, only the life extension from the latest food counts. For instance, buying 1-day food after 7-day food reduces the life extension to 1 day instead of 7, resulting in a loss of value.

```
126    function buyFood(uint256[] memory tokenIds, uint256 foodWorkTime)
127    external
128    payable
129 {
130    uint256 foodPrice = _foodPrices[foodWorkTime];
131    if (foodPrice <= 0) {
132        revert FoodNotOnSale(foodWorkTime);
133    }
134    uint256 foodPriceSum = 0;
135    uint256 currentTime = block.timestamp;
136    for (uint256 i = 0; i < tokenIds.length; i++) {
137        address owner = _ownerOf(tokenIds[i]);
138        if (msg.sender != owner) {
139            revert MustBeNftOwner(owner);
140        }
141        foodPriceSum += foodPrice;
142        _lifePool[tokenIds[i]].workEndTime = uint64(
143            currentTime + foodWorkTime
144        );
145            emit FeedEvent(tokenIds[i], currentTime, foodWorkTime);
146        }
147        if (msg.value < foodPriceSum) {
148            revert EtherNotEnough(foodPriceSum);
149        }
150    }
```

Listing 2.4: src/Life.sol

Impact Users will get a shorter workEndTime than expected.

Suggestion Revise the logic to update workEndTime accordingly.

Feedback from the Project This is by design.

2.2.4 Lack of Refund in Function `buyFood()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `buyFood()` of the contract Life, there is no refund logic when the user pays more than needed.

```
126   function buyFood(uint256[] memory tokenIds, uint256 foodWorkTime)
127   external
128   payable
129{
130   uint256 foodPrice = _foodPrices[foodWorkTime];
131   if (foodPrice <= 0) {
132       revert FoodNotOnSale(foodWorkTime);
133   }
134   uint256 foodPriceSum = 0;
135   uint256 currentTime = block.timestamp;
136   for (uint256 i = 0; i < tokenIds.length; i++) {
137       address owner = _ownerOf(tokenIds[i]);
138       if (msg.sender != owner) {
139           revert MustBeNftOwner(owner);
140       }
141       foodPriceSum += foodPrice;
142       _lifePool[tokenIds[i]].workEndTime = uint64(
143           currentTime + foodWorkTime
144       );
145
146
147       emit FeedEvent(tokenIds[i], currentTime, foodWorkTime);
148   }
149   if (msg.value < foodPriceSum) {
150       revert EtherNotEnough(foodPriceSum);
151   }
152}
```

Listing 2.5: src/Life.sol

Impact Users cannot receive refunds.

Suggestion Implement a refund mechanism in the function `buyFood()`.

Feedback from the Project This is by design.

2.2.5 Lack of Check in Function `addNewAuction()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `addNewAuction()`, the owner should ensure `maxSellable_` is less than or equal to 511 and `startTokenID_` is larger than `cellMintedNum`.

```
244   function addNewAuction(
245       int256 targetPrice_,
246       int256 priceDecayPercent_,
247       int256 perTimeUnit_,
248       uint256 startTime_,
249       uint256 maxSellable_,
250       uint256 startTokenID_,
251       uint256 updateInterval_
252   ) public onlyOwner {
253       require(
254           _currentCellAuction.sold == _currentCellAuction.maxSellable,
255           "auction ongoing"
256       );
257       require(startTime_ > block.timestamp, "invalid startTime");
258       int256 decayConstant = wadLn(1e18 - priceDecayPercent_);
259       require(decayConstant < 0, "NON_NEGATIVE_DECAY_CONSTANT");
260       _currentCellAuction.startTime = startTime_;
261       _currentCellAuction.targetPrice = targetPrice_;
262       _currentCellAuction.decayConstant = decayConstant;
263       _currentCellAuction.perTimeUnit = perTimeUnit_;
264       _currentCellAuction.maxSellable = maxSellable_;
265       _currentCellAuction.startTokenID = startTokenID_;
266       _currentCellAuction.updateInterval = updateInterval_;
267   }
```

Listing 2.6: src/CellGame.sol

Impact First, if `startTokenID` is set as an incorrect value, users won't be able to mint a new cell since the target `tokenId`, which is calculated via `startTokenID`, has already been minted. Second, if `maxSellable_` is larger than 511, users can get two cells in the same random number in one round, which is inconsistent with our design.

Suggestion Add relevant check in function `addNewAuction()`.

2.2.6 Lack of Interface to Withdraw `_poolFeeCollected` Fee

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `createLife()`, 70% of the cell rent fee is allocated to the cell's owner, 10% to the dev team, and 20% to a liquidity pool. The project implements an interface called `withdrawRentFee()` for the cell's owner. For the dev fee, the project implements an interface called `withdrawDevFee()`. However, there is no interface for the liquidity pool.

```
163   function createLife(uint256[][] memory cellsPositions_) public payable {
164       require(
165           cellsPositions_.length >= 2 && cellsPositions_.length <= 9,
166           "can only use 2-9 cells!"
167       );
168       uint256 cumulatedPrice = 0;
```

```
169         uint256[] memory cellGenes = new uint256[](cellsPositions_.length);
170         uint32[] memory livingCellTotals = new uint32[](cellsPositions_.length);
171 
172 
173         uint256 totalRentFeeCollected = 0;
174         for (uint256 i = 0; i < cellsPositions_.length; i++) {
175             uint256 tokenId = cellsPositions_[i][0];
176             CellGene storage cellGene = _cellPool[tokenId];
177             uint256 absoluteTimeSinceStart = block.timestamp -
178                 cellGene.bornTime;
179             uint256 cellRentPrice = getCellRentPrice(
180                 cellGene.rentedCount,
181                 absoluteTimeSinceStart
182             );
183 
184 
185             cellGenes[i] = cellGene.bitmap.getBucket(0);
186             livingCellTotals[i] = cellGene.livingCellTotal;
187             cellGene.rentedCount += 1;
188 
189 
190             uint256 rentFee = (cellRentPrice * 70) / 100;
191             _rentFeeCollected[tokenId] += rentFee;
192             totalRentFeeCollected += rentFee;
193 
194 
195             emit MintFeeReceived(tokenId, rentFee);
196 
197 
198             cumulatedPrice += cellRentPrice;
199         }
200         uint256 remainFee = cumulatedPrice - totalRentFeeCollected;
201         _devFeeCollected += remainFee / 3;
202         _poolFeeCollected += remainFee - remainFee / 3;
203 
204 
205         emit MintFeeForDevReceived(remainFee / 3);
206         emit MintFeeForPoolReceived(remainFee - remainFee / 3);
207 
208 
209         require(msg.value >= cumulatedPrice, "Insufficient funds");
210 
211 
212         _life.createLife(
213             msg.sender,
214             cumulatedPrice,
215             cellsPositions_,
216             cellGenes,
217             livingCellTotals
218         );
219 
220 
221         if (msg.value > cumulatedPrice) {
```

```
222             (bool sent, ) = payable(msg.sender).call{
223                 value: msg.value - cumulatedPrice
224             }(""); // Returns false on failure
225             require(sent, "failed to return Ether");
226         }
227     }
```

Listing 2.7: src/CellGame.sol

Impact The `_poolFeeCollected` fee cannot be withdrawn.

Suggestion Implement interface to withdraw the `_poolFeeCollected` fee.

2.2.7 Lack of Upper Limit in Function `Withdraw()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `CellGame`, the function `withdraw()` can get any amount from the contract. However, the owner is not expected to withdraw the cells' rent fee and pool's fee directly, which can be a centralized problem.

```
157     function withdraw(uint256 amount) public onlyOwner {
158         require(amount <= address(this).balance, "Insufficient balance");
159         address payable owner = payable(owner());
160         owner.transfer(amount);
161     }
```

Listing 2.8: src/CellGame.sol

Impact The owner can withdraw assets that do not belong to them.

Suggestion Add an upper limit check in function `withdraw()`.

2.3 Additional Recommendation

2.3.1 Lack of Check in Function `createLife()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `createLife()` only ensures that the `position` is less than 10. A zero value `position` will cause an underflow and result in a revert.

```
 96     function createLife(
 97         address to,
 98         uint256 bornPrice,
 99         uint256[][] calldata cellsPositions,
100         uint256[] calldata cellGenes,
101         uint32[] calldata livingCellTotals
102         ) external onlyCell {
103         for (uint256 i = 0; i < cellsPositions.length; i++) {
```

```
104         require(cellsPositions[i][1] < 10, "position error");
105       //...
106       for (uint256 i = 0; i < cellsPositions.length; i++) {
107         uint256 parentTokenID = cellsPositions[i][0];
108         newLife.parentTokenIds.push(parentTokenID);
109         uint256 position = cellsPositions[i][1];
110         uint256 x = ((position - 1) % 3) * 3;
```

Listing 2.9: src/Life.sol

Suggestion Add a zero check on `cellsPosition[i][1]`.

2.3.2 Incorrect Comments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The comments "withdraw eth from the contract" and "Obtain 512 unique random numbers for 10 rounds" are incorrect.

```
156   //withdraw eth from the contract
157   function withdraw(uint256 amount) public onlyOwner {
```

Listing 2.10: src/CellGame.sol

```
387   //Obtain 512 unique random numbers for 10 rounds
388   function getRandomNumber() public returns (uint256) {
```

Listing 2.11: src/CellGame.sol

Suggestion Revise the comments.

2.3.3 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `getTargetSaleTimeLogistic()` and library `SignedWadMath` are redundant.

```
45  function getTargetSaleTimeLogistic(
46      int256 sold,
47      int256 logisticLimit,
48      int256 timeScale
49  ) internal pure returns (int256) {
50      unchecked {
51          return
52              -unsafeWadDiv(
53                  wadLn(
54                      unsafeDiv(logisticLimit * 2e18, sold + logisticLimit) -
55                          1e18
56                  ),
57                  timeScale
```

```
58              );
59      }
60  }
```

Listing 2.12: src/lib/VRGDA.sol

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity >=0.8.0;
3
4   // ...
5   function toHoursWadUnsafe(uint256 x) pure returns (int256 r) {
6       assembly {
7           // Multiply x by 1e18 and then divide it by 3600.
8           r := div(mul(x, 1000000000000000000), 3600)
9       }
10  }
11
12  function fromHoursWadUnsafe(int256 x) pure returns (uint256 r) {
13      assembly {
14          // Multiply x by 3600 and then divide it by 1e18.
15          r := div(mul(x, 3600), 1000000000000000000)
16      }
17  }
```

Listing 2.13: src/lib/SignedWadMath.sol

Suggestion Remove the redundant code.

2.3.4 Improper usage of function Transfer

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `withdraw()` uses `transfer()` to transfer native token to the owner, which is not suggested.

```
394 // withdraw available balance from the contract
395 function withdraw(uint256 amount) public onlyOwner {
396     require(_withdrawable, "withdraw paused");
397     uint256 withdrawable = address(this).balance -
398         (_totalRentFee + _devFeeCollected + _poolFeeCollected);
399     require(amount <= withdrawable, "Insufficient balance");
400     address payable owner = payable(owner());
401     owner.transfer(amount);
402 }
```

Listing 2.14: src/CellGame.sol

Suggestion Use `call{value:amount}` or Openzeppelin's `sendValue`.

2.4 Note

2.4.1 Higher Cell Price Due to Round Down Design

Description In the function `getCellRentPrice()`, it always rounds down the `timeSinceStart` parameter in favor of the cells' owners and dev team. Thus, the returned cell price is higher than the theoretical price.

```
95  function getCellRentPrice(
96      uint256 rentedCount,
97      uint256 absoluteTimeSinceStart
98  ) public view returns (uint256) {
99      return
100         VRGDA.getVRGDAPrice(
101             toDaysWadUnsafe(
102                 absoluteTimeSinceStart -
103                     (absoluteTimeSinceStart %
104                         _lifeCreationConfig.updateInterval)
105             ),
106             _lifeCreationConfig.cellTargetRentPrice,
107             _lifeCreationConfig.decayConstant,
108             // Theoretically calling toWadUnsafe with sold can silently overflow but under
109             // any reasonable circumstance it will never be large enough. We use sold + 1 as
110             // the VRGDA formula's n param represents the nth token and sold is the n-1th token
                .
111             VRGDA.getTargetSaleTimeLogisticToLinear(
112                 toWadUnsafe(rentedCount + 1),
113                 _lifeCreationConfig.soldBySwitch,
114                 _lifeCreationConfig.switchTime,
115                 _lifeCreationConfig.logisticLimit,
116                 _lifeCreationConfig.timeScale,
117                 _lifeCreationConfig.perTimeUnit
118             )
119         );
120 }
```

Listing 2.15: src/CellGame.sol

2.4.2 Inconsistent `BLOCK_TIME`

Description The project sets `BLOCK_TIME` as 2 seconds, which is not consistent with the opbnb chain (1 second) and the BSC chain (3 seconds). Inconsistent block time will lead to a higher or lower evolution speed than designed.

Feedback from the Project The depolyer will set `BLOCK_TIME` before deploying the contract.

2.4.3 Lack of Access Control in `sendClaimEnergyRequest()`

Description Anyone can invoke the function `sendClaimEnergyRequest()` to update the claimTime.

```
8   function sendClaimEnergyRequest() public {
9       uint256 claimTime = block.timestamp;
10      emit ClaimEnergy(msg.sender, claimTime);
```

```
11    }
```

Listing 2.16: src/Energy.sol

Feedback from the Project This functionality is not implemented yet.

2.4.4 Lack of Evolution Implementation

Description According to the documentation[1], the cell and life will evolve according to the environment (e.g., land, air, sea) and time. Furthermore, cells should enter a cool-down period after synthesizing a life form. However, there is no implementation for these evolution-related functions.

Feedback from the Project This functionality is not implemented yet.

[1] https://cellulalifegame.gitbook.io/cellula/gameplay


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS