Kugle Security Review Report

Version 1.0

Strapontin & x0t0wt1w

August 29, 2025

Kugle Security Review Report

Strapontin, x0t0wt1w

Prepared by: Strapontin

Lead Auditors:

- Strapontin
- x0t0wt1w

Table of Contents

- Table of Contents
- Kugle Summary
- Disclaimer
- Risk Classification
- Security Review Details
  - Scope
- Executive Summary
  - Issues found
- High
- Medium
- Low
- Informational

Kugle Summary

Kugle is a platformer mobile blockchain game featuring virtual pets. On the blockchain, players are able to buy, hatch and breed kugles to increase their collection, as well as stake them to receive rewards. There are three different tokens that users will be able to use in the game: Gu, Carbon, and Heat. Gu has a fixed total supply and will be bought and staked by users to receive rewards or advantages, such as being able to stake more kugles. Carbon and Heat can be earned by staking owned kugles. While they have a fixed total supply in the audited version of the code, these tokens should become elastic in a future version to allow users to earn a more consistent amount when staking a kugle. Heat and Carbon can be spent to hatch kugles and breed them together.

Disclaimer

The Strapontin and x0t0wt1w team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security review by the team is not an endorsement of the underlying business or product. The security review was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts. A security review does not prove that the code is bug-free. Gas and performance issues reported below will be grouped in the Informational category.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H	M
Likelihood	Medium	H	M	L
	Low	M	L	L

We use the pashov severity matrix to determine severity. See the documentation for more details.

Security Review Details

Review commit hash : `341a21c70e2daa8a43bdd62d352c1c157eb7cd32`

Scope

```
src/
|-- NFT/
|    |-- Breeding.sol
|    |-- Hatch.sol
|    |-- IKugleNFT.sol
|    |-- KugleFactory.sol
|    |-- KugleNFT.sol
|    |-- Rewards.sol
|    |-- Staking.sol
|-- Seller/
|    |-- IBoosterSeller.sol
|    |-- KugleBoosterSeller.sol
|-- Tokens/
|    |-- Carbon.sol
|    |-- Gu.sol
|    |-- Heat.sol
|    |-- IKugleToken.sol
|    |-- IReactor.sol
|    |-- KugleTokenFactory.sol
|    |-- Reactor.sol
```

Executive Summary

Issues found

In this security review, a total of 7 High, 3 Medium, 8 Low and 14 Info severity issues were found.

	High	Medium	Low	Info
Issues found	7	3	8	14

High

[H-01] Users are not encentivized to call `Breeding::breed` with `_priceAdditionnalCarbon` other than `0`.

Description:

The protocol allows users to breed 2 owned NFTs in order to create a new one. In the off-chain game, users can choose components of one of the parents to pass the gene onto the newly created NFT, at the expense of additional carbon.

Because the on-chain implementation of the breeding system is permission-less, and does not keep track of the genes to pass to the new NFT, users will never pass another value than 0 for the variable _priceAdditionnalCarbon because it will increase the cost of the breeding.

Impact:

Users will pay less than they should.

Recommended Mitigation:

This function implementation could be refactored in several ways, depending on what to achieve :

1 - **If you want to make this completely decentralized**, consider first adding the kugle's details on-chain (wings, color…). Then, remove the parameter `_priceAdditionnalCarbon` and add a new one with the genes that the users want to pass to the newly created NFT. You can then calculate the additional carbon to spend depending on the selected elements from the user. Be aware that the randomness to determine kugles characteristics should also be implemented on-chain, which will be time-consuming to add.

2 - **If you want to enforce no cheating**, while keeping changes in the function to a minimum, you can consider making this function callable only by an automated trusted entity, and thus be sure that the correct amount of carbon will be taken from the user. This would also allow you to reduce the size of the contract since most of the calculations can be done securely off-chain and would not be necessarily coded in the contract, like for example the `require` to assert that the kugles used are owned by the user.

3 - **If you want to keep changes to the lowest possible, and still let users breed tokens by themselves**, you can let the function as-is (just remove the parameter `_priceAdditionnalCarbon`) and calculate the carbon-debt of the user off-chain and add it later through admin-only function. While the debt is not paid, the user would not be able to breed (or even to unstake if you want to force pay-back) to avoid accumulation of debt.

Kugle team: Acknowledged. This value will be checked off-chain, and users who are cheating will not have the expected metadata for their NFT.

[H-02] Adding boosters to an edition that isn't the last one (using `addKuglesIds`) will result in a corruption of the booster

Description:

The edition struct contains a variable `uint256[2] kuglesIds`, where the 0-index of the array is the id of the first kugle from this edition, and the 1st index is the id of the last kugle of the edition.

This works well until we add a new kugle to an edition that isn't the last one. Since `_kuglesCount` was increased by other editions, the new value is now an incorrect value, taking into account all other editions created after the one we are editing.

Basically, the same thing is happening to `uint256[2] boosterIds`.

Impact:

Some view functions will return inacurrate values related to this edition. Users will be able to buy more boosters than they should.

Proof of Concept:

```
// forge test --mt testAddingKuglesToPreviousEditionBreaksCount
function testAddingKuglesToPreviousEditionBreaksCount() public {
    uint256 boosterPrice = 10_000;
    vm.deal(owner, 1 ether);

    vm.startPrank(owner);

    boosterSeller.createEdition("Edition1", boosterPrice, "Edition", 6,
        0, 0, 0);
    boosterSeller.createEdition("Edition2", boosterPrice, "Edition",
        30, 0, 0, 0);

    uint256 kuglesInEditionBefore = 6;
    uint256 boostersInEditionBefore = 2;

    assertEq(boosterSeller.editionKuglesLength(1),
        kuglesInEditionBefore);
    assertEq(boosterSeller.editionBoostersLength(1),
        boostersInEditionBefore);

    // Add only 3 kugles to Edition1
    boosterSeller.addKuglesIds(1, 3);

    // The code thinks there are now 36 more kugles in the edition
    assertEq(boosterSeller.editionKuglesLength(1),
        kuglesInEditionBefore + 33);
    assertEq(boosterSeller.editionBoostersLength(1),
        boostersInEditionBefore + 11);
    assertEq(boosterSeller.editionBoughtBoostersNumber(1), 0);

    // Even though the 1st edition has only 3 boosters, we can buy 13
    boosterSeller.buyBoosters{ value: boosterPrice * 13 }(1, 13, false,
        owner);

    vm.stopPrank();
}
```

Recommended Mitigation:

The way edition and kugleIds are implemented seems prone to errors. I would recommend changing this implementation entirely.

Using a counter for the amount of total and available boosters seems more efficient. The use of arrays for this are overkill.

If you need to know in which edition belongs a kugle, I would then recommend to use a mapping. You can then use the kugleIds available when buying a new pack, instead of pre-allocating an id for each pack.

Kugle team: Fixed in 5`b036c2492be99f22f4e5feef55ae0a42526e07f`.

[H-03] - DoS of `buyBoosters()` if the user stakes his booster to the address(0)

Description:

`buyBoosters()` allows a user to purchase a pack containing three NFTs, with the option to stake them immediately after purchase. However, no validation is done on the address provided for staking. This means that the user can choose to stake their NFTs to address 0x0, which leads to several issues:

- The system counts the boosters based on the buyer's address. However, 0x0 can be treated as a valid address in this context, causing the booster count to be inaccurate.
- Due to the incorrect counter, the function tries to `mint()` a `tokenId` that has already been assigned. This `tokenId` is already owned by the `KugleBoosterSeller` contract, making the mint impossible and triggering a revert.
- The error prevents any user from purchasing boosters again.

Impact:

DoS of `buyBoosters()`.

Proof of concept:

```
//forge test --mt test_BuyBoosterAndStakeToZeroAddress_
function test_BuyBoosterAndStakeToZeroAddress_() public {
        vm.startPrank(owner);
        createEdition();
        boosterSeller.setEditionAvailability(1, true);
        boosterSeller.setEditionVisibility(1, true);
        vm.stopPrank();

        vm.deal(randomUser, 1e7);

        vm.startPrank(randomUser);
        boosterSeller.buyBoosters{ value: 10_000 }(1, 1, true, address
            (0));
        uint256[] memory tokens = kugleNFT.stakedTokenOfOwner(address
            (0));
        assertEq(tokens.length, 3);

        //User tries to buy a new booster
        vm.expectRevert();
        boosterSeller.buyBoosters{ value: 10_000 }(1, 1, false,
            randomUser);
        vm.stopPrank();
    }
```

Recommended Mitigation:

Add the following require at the beginning of the function `buyBooster()`:

```
require(to != address(0), "Invalid address");
```

Kugle team: Fixed in commit `da489d0969ac5a1fc0cdfbd3324718cc222790e5`.

[H-04] `ClaimRewards()` earns `1e18` times more rewards and exceed the NFT reward thresholds

Description:

The `claimReward()` function is defined in several main stages. Firstly, it calculates the maximum rewards for each token with `calculateMaxRewards()`, which then calls `calculateMaxRewardOfKugle()`.

In this calculation, the rewards have already been multiplied by 1e18 to match the correct number of tokens.

```
uint256 totalRewardPotential = ((elapsedDays - lastClaimAfter) * ((
    _rewardsInfos.dailyCarbonPerKugle * _rewardsInfos.
    assignedCarbonRewards) / 100) ) * 10 ** 18;

uint256 totalRewardPotential = ((elapsedDays - lastClaimAfter) * ((
    _rewardsInfos.dailyHeatPerKugle * _rewardsInfos.assignedHeatRewards)
     / 100) ) * 10 ** 18;
```

Secondly, after checking that the user is not requesting too many rewards, the function calls `realValueRewards()`. This will calculate the actual rewards, taking into account NFT's individual shares of Heat and Carbon. In addition, there is a check that each NFT reward does not exceed the thresholds defined by the protocol.

However, rewards are once again multiplied by 1e18:

```
uint256 heatRewards = _virtualHeat * 10 ** 18;
uint256 carbonRewards = _virtualCarbon * 10 ** 18;
```

Finally, the function transfers the actual rewards to the user and updates the various variables.

Impact:

`ClaimRewards()` returns rewards that are `1e18` times too high

Recommended Mitigation:

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Rewards.sol#L82-L83

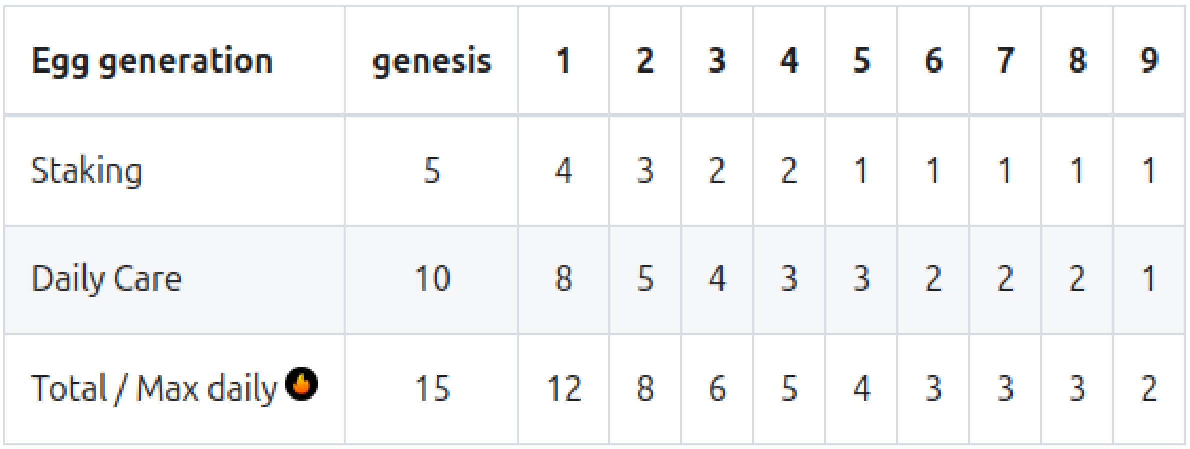
Delete the multiplication by 1e18. Use `claimReward()` instead of `calculateMaxRewardofKugle()` in `unstake()`, `breed()` and `hatch()`.

Kugle team: Fixed in commit `7f89fbbf24fab2f3dd5c3300bfe1e17f1ae02bde`.

[H-05] The calculations of rewards for staked kugles are wrong

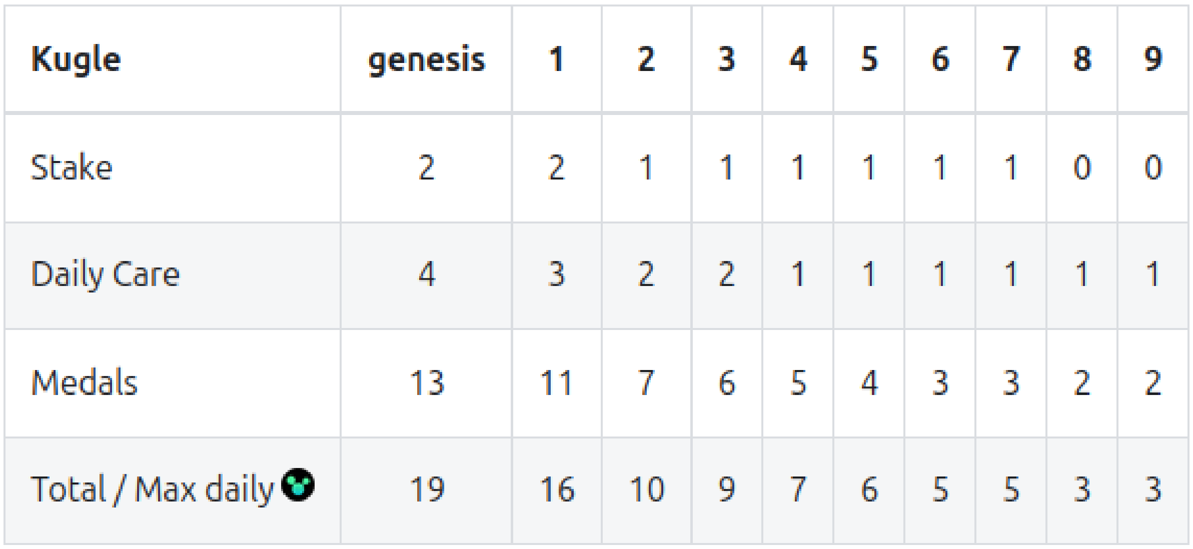
Description:

The documentation shows the following rewards for heat and carbon :



Egg generation	genesis	1	2	3	4	5	6	7	8	9
Staking	5	4	3	2	2	1	1	1	1	1
Daily Care	10	8	5	4	3	3	2	2	2	1
Total / Max daily 🔥	15	12	8	6	5	4	3	3	3	2

Figure 1: Heat rewards



Kugle	genesis	1	2	3	4	5	6	7	8	9
Stake	2	2	1	1	1	1	1	1	0	0
Daily Care	4	3	2	2	1	1	1	1	1	1
Medals	13	11	7	6	5	4	3	3	2	2
Total / Max daily	19	16	10	9	7	6	5	5	3	3

Figure 2: Carbon rewards

The calculations for rewards differ slightly depending on the kugles' generation. Between Heat and Carbon, the calculations are the same, with the appropriate variables :

*All calculations start from the base result of **totalRewardPotential** (TRP) :*

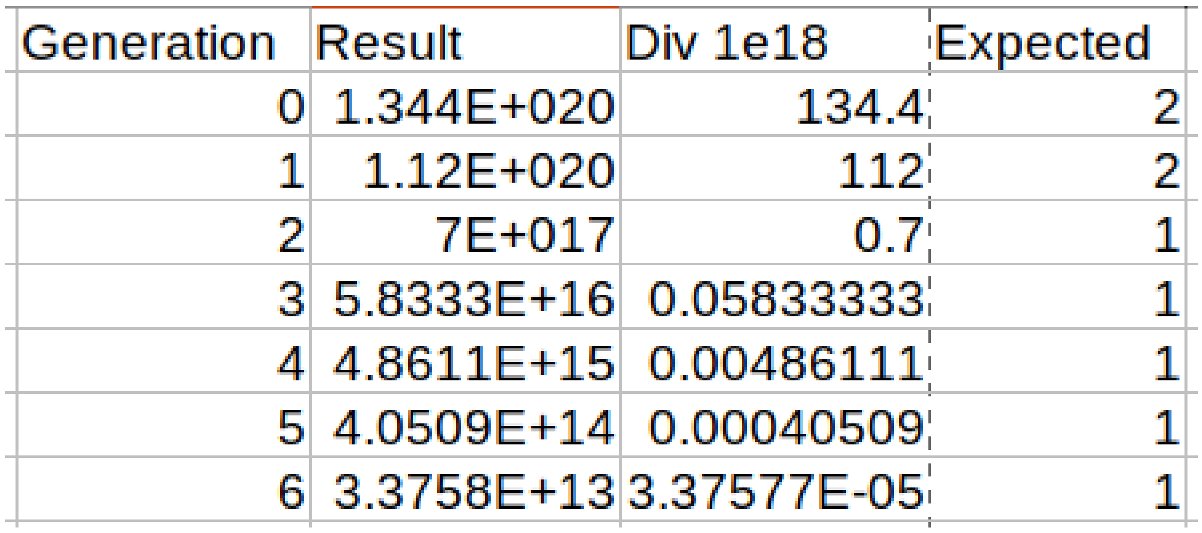
```
TRP = daysSinceLastClaim * dailyCarbonPerKugle / 100
```

```
Gen 0, maxRewards = TRP * rewardsMultiplier / 10^rewardsDecimals
```

```
Gen 1, maxRewards = TRP
```

```
Gen 2+, maxRewards = (((TRP / rewardsMultiplier^(generation-1)) / 4) *
    3) / 10^rewardsDecimals
```

The differences in those calculations creates a huge difference in the rewards earned depending on the generation, but also all of the results are off from what's expected from the documentation :



Generation	Result	Div 1e18	Expected
0	1.344E+020	134.4	2
1	1.12E+020	112	2
2	7E+017	0.7	1
3	5.8333E+16	0.05833333	1
4	4.8611E+15	0.00486111	1
5	4.0509E+14	0.00040509	1
6	3.3758E+13	3.37577E-05	1

Figure 3: Actual results and expected values

Impact:

Kugles from generation 0 & 1 earn more than 50x the amount of expected rewards. Kugles from generations 2+ earn less, which may be 0 due to precision loss.

Recommended Mitigation:

Update the reward function to have results according to the docs. Either find a more accurate calculation, or hard-code reward amount to be sure it is correct.

Also, the variable `delayInHours` reduces the reward amount as it rounds the time computation down to days instead of seconds. This means users will only have optimal earnings when calling this right after 24 hours have passed since they staked. Consider allowing users to earn rewards down to the second, so they don't loose anything.

Finally, note that the documentation says that subsequent generations earn 20% less resources, AND eggs resulting from breeding have their resources reduced by 25%. We believe that the second assertion is false and should be removed from the documentation

Kugle team: Fixed in commit 7`f89fbbf24fab2f3dd5c3300bfe1e17f1ae02bde`.

[H-06] Possible DoS due to Out Of Gas exception when looping through unbounded array

Description:

In `Staking::stakedTokenOfOwner`, we loop through all kugles as long as the address passed in parameter owns at least one. Since the amount of kugles is unbounded, this function will be very gas-costly, and may revert and prevent users to call `Reactor.unblockGu` since it calls `stakedTokenOfOwner`.

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Staking.sol#L114-L132

The same kind of code-looping structure is also implementing in `KugleNFT::tokenOfOwner`.

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/KugleNFT.sol#L19-L37

Impact:

Functions with high consumption of gas. May revert and DoS some functionalities (users can't unlock they GU).

Recommended Mitigation:

For `KugleNFT::tokenOfOwner`, this function could be easily replaced by an inheritance of ERC721Enumerable to reliably handle users by using the function `tokenOfOwnerByIndex` along with `balanceOf(user)` :

```
function newtokenOfOwner(address user) external view returns (uint256[]
     memory result) {
    uint256 tokenCount = balanceOf(user);

    for (uint256 i = 0; i < tokenCount; i++) {
        // tokenOfOwnerByIndex returns the token owned by user at index
             i
        result.push(tokenOfOwnerByIndex(user, i));
    }
}
```

For staked kugle, the implementation will be more tricky, since all staked kugles are owned by the contract. Since the mapping `_stakedKugles` takes a kugleId and returns staking details about that kugle, we can see that this mapping is very efficient when we know the kugleId to get the details from. However, since we have fetch all staked kugles of a user, we can see that using this variable for this behaviour is not efficient.

I suggest creating a new mapping that will record owned staked kugles ids for more efficiency when reading all staked kugles of an owner :

```
mapping(address user => uint256[]) userStakedKugleId;

// Modify the already existing function
function stakedTokenOfOwner(address user) external view returns (
    uint256[] memory) {
    return userStakedKugleId[user];
}
```

In order for `userStakedKugleId` to hold the correct values, we need to update the staking and unstaking of kugles :

```
function _stake(uint256 _kugleId, address _owner) internal {
    require(userCanStake(), "p");
    transferFrom(msg.sender, address(this), _kugleId);
    StakeInfo memory stakedKugle = StakeInfo({
        owner: _owner,
        stakingDate: block.timestamp,
        lastRewardClaimDate: block.timestamp,
        kugleId: _kugleId
    });

    _stakedKugles[_kugleId] = stakedKugle;
    _nbStakedKugles[_owner]++;
    _totalStakedKugles++;

+   // Adds kugles id to the mapping-array
+   userStakedKugleId[_owner].push(_kugleId);

    emit StartStaking(_owner, _kugleId);
}

+ // Remove a specific id from the mapping array
+ function unstakeUsersKugle(uint256 _kugleId) public {
+     uint256[] storage kugleIds = userStakedKugleId[msg.sender];
+
+     for (uint256 i = 0; i < kugleIds.length; i++) {
+         if (kugleIds[i] == _kugleId) {
+             kugleIds[i] = kugleIds[kugleIds.length - 1]; // Move the
    last element to the deleted spot
+             kugleIds.pop(); // Remove the last element
+             break; // Saves gas
+         }
+     }
+ }
```

Don't forget to also add the call to the function `unstakeUsersKugle` when unstaking kugles

Kugle team: Fixed in commit `0a853ed07018eef794722ec4e463ff5e95da48c4`.

[H-07] It is possible to use the same kugle for breeding

Description:

The game allows user to use 2 hatched, fertile kugles to reproduce and create a new NFT of next generation. However, there is no check for the parameters of the kugle used, allowing a user to use the same kugle as both parents to create a new one.

This can also be exploited to use a kugle for more breeding than their max intended fertility. While exploiting the breeding action this way would make the same kugle level increase twice, a kugle from almost the max level would be able to level up higher than their max fertility level.

Impact:

Abuse of the function to create more kugles than intended, reducing income to the protocol.

Proof of Concept:

Add this test in `kugleNft.t.sol`:

```
// forge test --mt testBreedingSameKugle
function testBreedingSameKugle() public {
    vm.startPrank(owner);

    uint256[] memory toMint = new uint256[](1);
    toMint[0] = 1;
    kugleNFT.mintKuglesTo(toMint, address(randomUser));
    kugleNFT.setKugleLevel(1, 2);
    vm.stopPrank();

    vm.startPrank(randomUser);

    carbonToken.approve(address(kugleNFT), type(uint256).max);
    kugleNFT.breed(1, 1, 0, false, 0);

    vm.stopPrank();
}
```

Recommended Mitigation:

Add a condition at the beginning of the function to be sure both kugles used are not the same.

Kugle team: Fixed in commit `9ae65689637e980f9b86dd7b6c396711a5d99746`.

Medium

[M-01] Precision loss when using division before multiplication

Description:

Solidity types don't handle decimals. In a same operation, doing a division before a multiplication may result in a loss of precision. Multiplications should always be done first when applicable.

Example :

```
uint256 a = 111;
```

```
uint256 b = 1000;

uint mistake = a / b * b; // This calculation will return 0
uint correct = a * b / b; // This calculation will return 111
```

Occurrences:

`Rewards::calculateMaxRewardOfKugle` : Dividing by 100 before multiplying by `1e18` may loose some precision. Consider removing the division and multiplying by `1e16` instead :

```
// Before
uint256 totalRewardPotential = (
    (elapsedDays - lastClaimAfter)
        * ((_rewardsInfos.dailyCarbonPerKugle * _rewardsInfos.
           assignedCarbonRewards) / 100)
) * 10 ** 18;

// After
uint256 totalRewardPotential = (
    (elapsedDays - lastClaimAfter)
        * _rewardsInfos.dailyCarbonPerKugle * _rewardsInfos.
           assignedCarbonRewards
) * 10 ** 16;
```

`Rewards::realValueRewards` : Multiplying by the bank part after dividing by the max capacity per kugle for both heat and carbon will lose precision.

```
// Current
heatReal = (heatRewards / (_rewardsInfos.maxCapacityHeatPerKugle * eggs
    )) * (bankPartHeat * eggs);
carbonReal = (carbonRewards / (_rewardsInfos.maxCapacityCarbonPerKugle
    * kugles)) * (bankPartCarbon * kugles);

// With no precision loss
heatReal = heatRewards * bankPartHeat / _rewardsInfos.
    maxCapacityHeatPerKugle;
carbonReal = carbonRewards * bankPartCarbon / _rewardsInfos.
    maxCapacityCarbonPerKugle;
```

Kugle team: Fixed in commit `3037155281b07d0589ec2210819f7285f4947f51` & `b16882c498fe98ad589afe5bb4e96ddbdcb54d94`.

[M-02] Tiers in guRewardsTiers are not arranged in ascending order of guHeld leading to a wrong popcap

Description:

In the game, we have the following structure:

```
struct GuRewardTier {
    uint256 guHeld; // Number of GUs the user must block to reach
                    // this tier
    uint256 popCap; // Maximum number of NFTs the user can stake}
```

We also have an array `_guRewardsTiers`, which contains elements of type `GuRewardTier`. This array is used to define different reward tiers based on the number of GUs blocked by the user in Reactor.sol.

The `addOrDeleteGuRewardTier()` function can be used to add or remove a tier. However, when a tier is added, it is systematically inserted at the end of the `_guRewardsTiers` array. This poses a problem if the added tier does not correspond to the largest number of blocked GUs, as the order of the array is no longer ascending.

In `PopCapNotReached()` and `userCanStake()`, `popCap` is calculated on the assumption that the tiers in `_guRewardsTiers` are sorted in ascending order. If this is not the case, errors may occur.

In the loop, to determine the tier to which the user belongs, we check that the Gus blocked by the user are between 2 tiers, but the tiers are not sorted in ascending order so the function's return is incorrect.

In addition, the last check of `PopCapNotReached()` assumes that the last element of `_guRewardTiers` is the one corresponding to the highest number of blocked GUs.

```
if (guBlocked >= guRewardTiers[size - 1].guHeld) {
    popCap = guRewardTiers[size - 1].popCap;
}
```

If `_guRewardsTiers` is not sorted correctly, then `popCap` may be incorrectly defined.

Error example:

Let's say we have the following tiers : `(10.3), (15.5), (30.8)`

The owner adds a new tier with `guHeld` = 5 and `popCap` = 2 by calling `addOrDeleteGuRewardTier()`.

Instead of inserting the tier in the correct order, it is added at the end of the array, giving : `(10,3), (15,5), (30,8), (5,2)`

A user who has blocked 30 Gus or more should reach the Tier (30,8) and be able to stake 8 NFTs, but due to the poor sorting of the array, the last condition of `PopCapNotReached()` will set that the user is in the Tier (5,2) and will only be able to stake 2 NFTs.

Impact:

User can't stake as much Kugles as he has the right to. User can't unblock/withdraw a certain number of his Gus.

Recommended Mitigation:

When adding a tier using `addOrDeleteGuRewardTier()`, ensure that the new tier is inserted at the correct position in the array to maintain the ascending order of tiers based on `guHeld`:

```
function addOrDeleteGuRewardTier(uint256 index, uint256 _guHeld,
    uint256 _popCap)
     external
     onlyOwner
{
     if (_popCap == 0) {
         require(index < _guRewardTiers.length && index >= 0, "Index out
             of bounds");
         for (uint256 i = index; i < _guRewardTiers.length - 1; i++) {
             _guRewardTiers[i] = _guRewardTiers[i + 1];
         }
         _guRewardTiers.pop();
     } else {
         // Create new GuRewardTier
         GuRewardTier memory newTier = GuRewardTier(_guHeld, _popCap);
         uint256 length = _guRewardTiers.length;

         // If empty array or new _guHeld greater than the last, add at
            the end
         if (length == 0 || _guHeld > _guRewardTiers[length - 1].guHeld)
             {
             _guRewardTiers.push(newTier);
         } else {
             // Finding the insertion position
             uint256 i;
             for (i = 0; i < length; i++) {
                 if (_guHeld <= _guRewardTiers[i].guHeld) {
                     break;
                 }
             }
             // Increase size and offset elements
             // Add a space
             _guRewardTiers.push(_guRewardTiers[length - 1]);
             for (uint256 j = length - 1; j > i; j--) {
                 _guRewardTiers[j] = _guRewardTiers[j - 1];
             }
             _guRewardTiers[i] = newTier; // Insert at position found
         }
     }
}
```

Note: Delete `setGuRewardTier()` to avoid breaking the tier order.

Kugle team: Fixed at 69a0d30327e294eb3f49ae54064645ed923edea2 and 4c68d90bf03e6c5880572cf117f245975a1836

[M-03] - `setTokenId()` function leads to incorrect tracking of the number of NFTs.

Description:

The `setTokenId()` function is used to modify the counter that tracks the total number of NFTs created in the game.

This counter is crucial to ensure consistency in IDs and NFT tracking.

Impact:

The function poses several risks to the integrity of the game.

First, it can lead to inconsistencies in ID tracking: if the counter is adjusted without creating or destroying NFTs, the IDs of future NFTs will no longer match the expected sequence, potentially causing duplicates or gaps in the numbering. This could cause players to lose confidence in the reliability of the system.

Additionally, this function can falsify the total number of NFTs, misleading players about the actual quantity and rarity of these assets, which in turn affects their perception of the game's value and rules.

Using this function would not allow correcting any mistakes, but would even open doors to compromising the creation of new kugles, and existing ones.

In summary, allowing such manual modification introduces a significant risk of desynchronization between the IDs and the total number of NFTs, compromising the game's integrity and exposing the system to critical errors.

Recommended Mitigation:

Delete the `setTokenId()` function.

Kugle team: Fixed in commit 7`c4c5a33caf8a61ee9fed605ca6997f0a6c10311`.

Low

[L-01] User can't hatch more than 255 kugles at the same time

Description:

The hatching function verifies that all kugles passed as parameters are owned by `msg.sender`. However, the parameter is a `uint16[]` while the for loop uses a `uint8`.

https://github.com/KugleCorp/kugle_contracts_foundry/blob/cfd23dfb2d1d6e3e213a184cf64dfd99db281b28/src/NFT
L14C19

```
function hatch(
    uint256[] calldata _kugleIds,
    bool _stakeKugle
) external {
@>  (uint8 i = 0; i < _kugleIds.length; i++) {
        require(_kugleLevel[_kugleIds[i]] == 1, AlreadyHatch(_kugleIds[
            i]));
        require(
            ownerOf(_kugleIds[i]) == msg.sender
                || _stakedKugles[_kugleIds[i]].owner == msg.sender,
            NotOwned(_kugleIds[i], msg.sender)
        );
    }
```

Mitigation:

Use `uint256` to allow users to hatch as many eggs as they want.

Kugle team: Fixed in commit 8309a6a40e2633c3905879db1468de8615642775.

[L-02] `percentageToKeepForOwner` doesn't enforce the correct amount of NFTs send to the owner

Description:

The `edition` struct contains a variable `percentageToKeepForOwner` for the amount of packs that must go toward the owner. However, this assertion is not coded correctly and leads to 2 issues.

1. Once an edition is available and visible, users can buy all the packs from this edition even if the owner did not claim any.
2. If users buy boosters before the owner has claimed all of theirs, then they will count toward the boosters bought by the owner when calculating how many boosters the owner bought, and they will be able to claim less then.

Impact:

The owner will not get as many packs as they are supposed to.

Proof of Concept:

```
// forge test --mt testEditionPercentageForOwner
function testEditionPercentageForOwner() public {
    uint256 boosterPrice = 10_000;
    uint256 amountOfBoosters = 10;

    vm.deal(randomUser, 1 ether);

    vm.startPrank(owner);

    // Creates a simple edition with 100% boosters for the owner
    boosterSeller.createEdition(
        "Edition1", boosterPrice, "Edition", 3 * amountOfBoosters, 100,
            0, 0
    );

    boosterSeller.setEditionVisibility(1, true);
    boosterSeller.setEditionAvailability(1, true);

    vm.stopPrank();

    // A user can buy packs, but he should not
    vm.prank(randomUser);
    boosterSeller.buyBoosters{ value: boosterPrice }(1, 1, false,
       randomUser);
}
```

Recommended Mitigation:

When creating a new edition, mint at the same time all boosters that the owner should have.

If this is too much gas-consuming for one transaction, then consider preventing the change of edition availability to true while the owner has not claimed all their packs.

Kugle team: Fixed in commit 9ae578eb0d148b78399806139898093d85cca5ba.

[L-03] Missing input checks in `setRewardsInfos()` could lead to no rewards for the users

Description:

The `setRewardsInfos()` function in KugleFactory.sol has no check on inputs. In the function `calculateMaxRewardOfKugle()` of Rewards.sol, we have the following 2 calculations for `totalRewardPotential()`:

```
uint256 totalRewardPotential = ((elapsedDays - lastClaimAfter) * ((
    _rewardsInfos.dailyCarbonPerKugle * _rewardsInfos.
    assignedCarbonRewards) / 100)) * 10 ** 18;

uint256 totalRewardPotential = ((elapsedDays - lastClaimAfter) * ((
    _rewardsInfos.dailyHeatPerKugle * _rewardsInfos.assignedHeatRewards)
     / 100) ) * 10 ** 18;
```

However, due to the division by 100, we must ensure that `_rewardsInfos.dailyCarbonPerKugle * _rewardsInfos.assignedCarbonRewards` > 100 and that `_rewardsInfos.dailyHeatPerKugle * _rewardsInfos.assignedHeatRewards` > 100.

The opposite would imply a `totalRewardPotential` value of 0 and therefore no rewards for users.

Recommended Mitigation:

Add theses two lines in `setRewardsInfos()`:

```
require(_dailyHeatPerKugle * _assignedHeatRewards > 100, "Heat rewards
    too low");
require(_dailyCarbonPerKugle * _assignedCarbonRewards > 100, "Carbon
    rewards too low");
```

Kugle team: Refactored and outdated.

[L-04] Through misinput, an edition with empty `uniqueName` can be created

Description:

The creation of an edition requires to provide a `uniqueName` that is not already used by other editions.

However, this string can be left empty, and the edition would be created. There is no way to change this edition `uniqueName` after creation.

Impact:

An edition with no `uniqueName` can be created. It cannot be renamed.

Proof of Concept:

This test PASS without error.

```
// forge test --mt testCreateEditionEmptyUniqueName
function testCreateEditionEmptyUniqueName() public {
    string memory emptyUniqueName;

    vm.startPrank(owner);

    // Create a new edition with an empty `uniqueName`
    boosterSeller.createEdition(emptyUniqueName, 10000, "Edition", 33,
        0, 0, 0);

    // We can't change the edition's name because the code thinks the
        edition does not exist
    vm.expectRevert(abi.encodeWithSelector(EditionDoesNotExist.selector
        , 1));
    boosterSeller.renameEdition(1, "newName");
}
```

Recommended Mitigation:

When creating a new edition, be sure the `uniqueName` is not empty :

```
+ if (bytes(uniqueName).length == 0) revert WrongName(uniqueName);
```

Kugle team: Fixed in commit `c2fb1ea19d9a6d7847f713e1dee13fb8984ea77b`.

[L-05] `edition.percentageToKeepForOwner` will not be precise for edition with more than 100 boosters to sell

Description

Since solidity does not handle decimal for uints, this value will not be precise if the amount of boosters is greater than 100. For example, in an edition with 2000 boosters, there is no setting that allows the owner to get between 1001 and 1019 boosters, as 50% of 2000 is 1000, and 51% of 2000 is 1020.

If this precision is important to you, consider increasing the percentage precision by setting a number higher than 100.

Kugle team: Acknowledged.

[L-06] Royalties issues

Description:

Royalties with `ERC2981` used by `KugleFactory` must be set with `_setDefaultRoyalty` to be taken into accounts by marketplaces.

However, this is only done through calling `setRoyaltiesReceiverAddress`, while it should also be done when initializing the contract, and setting `_royaltiesRate` to a new value.

Impact:

The royalties will not be updated when the contract is created and when a new rate is set.

Recommended Mitigation:

Save the royalty receiver address to a new variable. Call `_setDefaultRoyalty` at the end of the `initialize` and `setRoyaltiesRate` :

```
+ address _royaltyReceiver;

function initialize() public initializer {
  // code omitted
  _royaltiesRate = 750;
+ _royaltyReceiver = msg.sender; // Maybe a different address to pass
    as parameter ?
}

function setRoyaltiesRate(uint16 _rate) external onlyOwner {
    _royaltiesRate = _rate;
+   _setDefaultRoyalty(_royaltyReceiver, _rate);

}

function setRoyaltiesReceiverAddress(address payable
    _royaltiesReceiverAddress)
    external
    onlyOwner
{
+   _royaltyReceiver = _royaltiesReceiverAddress;
    // IMPORTANT : si l'adresse change, cela ne sera actif que pour les
        NOUVEAUX kugle mint, pas de rétroactivité
    _setDefaultRoyalty(_royaltiesReceiverAddress, _royaltiesRate); //
       250 amount of royalties in % / 100
}
```

On another note, be aware that the comment in `setRoyaltiesReceiverAddress` is wrong. When you re-set the royalty address and call `_setDefaultRoyalty`, the royalties are set for ALL NFTs of this contract, whether if they are already created or not, as long as there aren't any royalty set for a specific token : https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/token/common/ERC2981Upgradeable.sol#L14-L15.

Kugle team: Fixed in comit `d82b9d5b61396954598c6f7f013926439ff662c1`.

[L-07] Users should claim all available carbon & heat by default instead of passing a value

Description:

In several places of the code, it is possible to claim carbon & heat staking rewards while doing another action. If the user specify 0 or a higher value than the max allowed to claim their reward, it is lost (some parts of the code actually revert if it is higher, but not all). If they claim their reward with a lower amount than the max available, they loose the difference.

Also, since the values passed are converted from integers to token decimals by multiplying by 1e18, users may miss on some decimals values of rewards.

Impact:

Users will loose value on their rewards. They may feel punished and loose faith in the project.

Proof of Concept:

Place this test in `kugleNft.t.sol`

```solidity
// forge test --mt testClaimManuallyIncurrLosses
function testClaimManuallyIncurrLosses() public {
    mintEggs();

    uint256[] memory kuglesIds = new uint256[](2);
    kuglesIds[0] = 1;
    kuglesIds[1] = 2;
    uint32[] memory kuglesIds32 = new uint32[](2);
    kuglesIds32[0] = 1;
    kuglesIds32[1] = 2;

    vm.startPrank(randomUser);

    kugleNFT.stake(kuglesIds);

    vm.warp(7 days);

    (uint256 maxHeatTokenClaimable,) = kugleNFT.calculateMaxRewards(
        kuglesIds32);
    console2.log(maxHeatTokenClaimable);

    uint256 balanceBefore = heatToken.balanceOf(randomUser);
    kugleNFT.claimReward(maxHeatTokenClaimable / 1e18, 0, kuglesIds32);
    uint256 balanceAfter = heatToken.balanceOf(randomUser);

    uint256 tokensEarned = balanceAfter - balanceBefore;
    uint256 loss = maxHeatTokenClaimable - tokensEarned;

    // By trying to earn the max claimable amount, the user lost 0.8
        tokens
    assertEq(loss, 0.8 * 1e18);

    vm.stopPrank();
}
```

Recommended Mitigation:

When a user can claim rewards, claim all rewards by default. If you end up in a scenario where users set a value for rewards, assume that it is already converted to the decimal value instead of multiplying by `10 ** 18`.

Kugle team: Ackknowledged. Users sometimes should earn a lower amount than the "max" calculated, depending on their activity in the game. The team will handle this when proposing the transaction through their UI.

[L-08] Approved users of Kugle NFT won't be able to use them the same way than if they were an owner

Description:

ERC721 implements an approval system to allow trusted addresses to handle NFTs on the owner's behalf. This trustness isn't implemented in the codebase, and approved users won't be able to perform actions in the protocol on behalf of the owner, like staking, breeding or hatching.

Impact:

Approved users can't act on behalf of the NFT owner.

Proof of Concept:

Add this test in `kugleNft.t.sol`:

```
    // forge test --mt testShouldAllowApprovedToStake
    function testShouldAllowApprovedToStake() public {
        address alice = makeAddr("alice");
        address bob = makeAddr("bob");

        vm.prank(owner);
        uint256[] memory toMint = new uint256[](1);
        kugleNFT.mintKuglesTo(toMint, alice);

        // Alice owns one kugle
        assert(kugleNFT.ownerOf(0) == alice);

        vm.prank(alice);
        kugleNFT.setApprovalForAll(bob, true);

        // Bob is approved for Alice's kugles
        assert(kugleNFT.isApprovedForAll(alice, bob) == true);

        // Bob cannot stake alice's kugle
        vm.expectRevert();
        vm.prank(bob);
        kugleNFT.stake(toMint);
    }
}
```

Recommended Mitigation:

When an approved user should be able to act on behalf of the owner, checks whether the `msg.sender` is either the owner, or if the owner approved them for all their NFTs or this particular NFT (`isApprovedForAll` & `getApproved`). This can be used using `_checkAuthorized`.

Occurences:

Breeding using authorized kugles

Hatching authorized kugles

Staking authorized kugles

Kugle team: Acknowledged.

Informational

[INFO-01] No function to update some edition properties.

Description:

An edition's struct has several properties, including `displayedName`, `percentageToKeepForOwner`, `heatToSendOnSale` and `carbonToSendOnSale`. Most of the other properties have a setting function, allowing for the owner to change it if needed.

However, the mentioned properties have no such functions. Consider adding these setters if you need to change the variable values after the contract's creation.

Kugle team: Fixed in `8e3f038c66b6b49e9036649791d442aa97d42b9e`.

[INFO-02] Hatching, Breeding and Unstaking should `claimRewards`

Description:

Through NFT staking, users will be able to receive rewards (carbon if nft level > 1, heat if nft level == 0). These rewards can be claimed by calling the following functions : - `Rewards::claimReward` - `Staking::unstake` - `Hatch::hatch` - `Breeding::Breed`

The issue is that all of these functions implement their own code for claiming the rewards. If sometimes change slightly, like in `unstake` where the user is "punished" if they try to claim more than the max allowed amount claimable, then this results in discrepency that may induce users to prefer one function over another.

This also adds drastically to the code size, readability, and may introduce subtle errors.

Recommanded mitigation:

It is recommended to have only one function to claim the rewards, like `Rewards::claimReward`. Then, other functions can either call this one directly or by using a modifier. This will reduce the code size and simplify the execution flow.

Kugle team: Fixed in commit `86e49d91cb6eefbc079df6607e37346e02cc8bf9`.

[INFO-03] `PopCapNotReached` & `userCanStake` could be more gas-efficient by looping backward

Description

The functions `Staking::userCanStake` and `Reactor::PopCapNotReached` basically implement the same logic. Consider grouping this in only one function to reduce code size and avoid potential errors when the code has small code differences, like in the cases of overlooked copy-paste.

This report will focus on `Staking::userCanStake`:

```
function userCanStake() internal view returns (bool) {
    uint256 guBlocked = guHeldContract.getBlockedGuByAddress(msg.sender
        );
    uint256 size = _guRewardTiers.length;
    uint256 popCap;
    for (uint256 i = 0; i < size - 1; i++) {
        if (guBlocked >= _guRewardTiers[i].guHeld && guBlocked <
            _guRewardTiers[i + 1].guHeld) {
            // @audit : This loops through the whole array with 2
                conditions in the if statement
            popCap = _guRewardTiers[i].popCap;
        }
    }
    // @audit : The last variable could be checked and returned before
        searching the entire array
    if (guBlocked >= _guRewardTiers[size - 1].guHeld) {
        popCap = _guRewardTiers[size - 1].popCap;
    }
    return _nbStakedKugles[msg.sender] + 1 <= popCap;
}
```

We see that this function could be refactored to be more efficient both gas-wise and for compiled size. The best refactoring case would be to loop backward from the reward tier array, which removes the second condition in the if statement, and return as soon as one array element meets the other condition, which both prevents looping through the entire array if a result is found before, and allows us to remove the second condition outside of the loop.

Mitigation example:

```
function userCanStake() public view returns (bool) {
    return _nbStakedKugles[msg.sender] + 1 <= getUserPopCap(msg.sender)
        ;
}

function getUserPopCap(address user) public view returns (uint256
    popCap) {
    uint256 guStaked = guHeldContract.getBlockedGuByAddress(user);
    GuRewardTier[] memory tiers = _guRewardTiers;

    for (uint256 i = tiers.length - 1; i >= 0; i--) {
        if (guStaked >= _guRewardTiers[i].guHeld) {
            return tiers[i].popCap;
        }
    }
}
```

These new functions can be tested with the following test added in `kugleNft.t.sol`:

```
// forge test --mt testUserCanStake
function testUserCanStake() public {
    uint256 stakingAmount = 5 ether;

    // Default value works well for user with no staking
    uint256 cap = kugleNFT.getUserPopCap(randomUser);
    assertEq(cap, 5);

    vm.prank(owner);
    kugleNFT.addOrDeleteGuRewardTier(1, stakingAmount, 8);

    vm.startPrank(randomUser);

    deal(address(guToken), randomUser, stakingAmount);
    guToken.approve(address(reactor), stakingAmount);
    reactor.blockGu(stakingAmount);

    // If user stakes enough, he has access to the new cap
    uint256 newCap = kugleNFT.getUserPopCap(randomUser);
    assertEq(newCap, 8);
}
```

Kugle team: Fixed in commit 2`d8b9c9da9b79b86a25999838145b3f293eec5a6`.

[INFO-04] Use call instead of transfer to transfer ETH

Description:

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/Seller/KugleBoosterSeller.sol#L211

The function `buyBoosters()` in KugleBoosterSeller.sol uses the `transfer()` to transfer eth.

You should not use `transfer()` as it imposes a limit of 2300 gas, which can cause failures if the recipient has complex logic. `call{value}()` is more flexible, doesn't limit gas and allows errors to be handled cleanly. It also avoids future problems related to Ethereum updates.

Recommended Mitigation :

Use `call()` instead of `transfer()`

```
(bool success, ) = payable(owner()).call{value: msg.value}("");
require(success, "Transfer failed");
```

Kugle team: Fixed in commit `eb418cb29a38cb671c78cb5f8d8a3b0ac1006567`.

[INFO-05] Missing checks for `address(0)`

Description:

The following functions do not check input addresses: - KugleFacotry.sol: `setContracts()` `setRoyaltiesReceiverAddress()` - KuggleBoosterSeller: `setKugleFactoryAddress()` `setHeatContractAddress()` `setCarbonContractAddress()` - Reactor.sol:`initialize()` - KugleTokenFactory: `unauthorizeContract()` `unauthorizeContract()` - Gu.sol: `initialize()`

Recommended Mitigation:

Add zero-address validation checks for all address parameters.

Kugle team: Fixed in commit `774551b95e1bdbb215f505fedd32eca10611a9ec`.

[INFO-06] A contract does not need to approve in order to send an NFT it owns

Description:

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Staking.sol#L90 If a contract owns an NFT, it doesn't need to call `this.approve(msg.sender, id)` before calling `transfer` (or `transferFrom`). This increases the size of the contract, and increase gas consumption.

Recommended Mitigation:

Delete the approval.

Kugle team: Fixed in commit `08f461afbfd4fc0047d404b1e9b10fdee11d8195`.

[INFO-07] Setting a variable to 0 after initialization is redundant

Description:

https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Rewards.sol#L85-L96
When creating a new variable, it is automatically set to 0. Setting it again is redundant.

Recommended Mitigation:

```
- if (eggs == 0) {
-     heatReal = 0;
- } else {
+ if (eggs != 0) {
      heatReal = (heatRewards / (_rewardsInfos.maxCapacityHeatPerKugle
          * eggs))
           * (bankPartHeat * eggs);
  }
- if (kugles == 0) {
-     carbonReal = 0;
- } else {
+ if (kugles != 0) {
      carbonReal = (carbonRewards / (_rewardsInfos.
          maxCapacityCarbonPerKugle * kugles))
           * (bankPartCarbon * kugles);
  }
```

Kugle team: Fixed.

[INFO-08] `require` is not needed when transferring Heat, Carbon & Gu

Description:

In several parts of the code, transfer of heat and carbon tokens are wrapped inside `require` . Since these tokens inherit from OpenZeppelin's ERC20Upgradeable contract, `transfer` and `transferFrom` will always return true, and revert when the transfer cannot be made : https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/token/ERC20/ERC20U

On the same note, the `require` to assert that the user has at least the required balance is not necessary either, because the transfer would revert if that is not the case : https://github.com/KugleCorp/kugle_contracts_foundry/

Idem for Gu transfer in `Reactor::blockGu` : https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src

Consider removing these requires to save some gas and reduce code's size.

Impact:

Increase of contract size, gas-inefficient function.

Recommended Mitigation:

Remove the requires. Keep only the tokens transfers. Occurrences :

```
// Breeding contract
// https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/
    NFT/Breeding.sol#L85
@>  require(carbonContract.balanceOf(msg.sender) + _carbonToClaim * 10
    ** 18 >= _price);

// https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/
    NFT/Breeding.sol#L91-L97
@>  require(
      amountToPay < 0
          ? carbonContract.transferFrom(
              address(carbonContract), msg.sender, uint256(amountToPay
                  * -1)
          )
          : carbonContract.transferFrom(msg.sender, address(
              carbonContract), uint256(amountToPay))
    );
```

```
// Hatch contract
// https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/
    NFT/Hatch.sol#L51-L56
@>      require(
            heatContract.transferFrom(msg.sender, address(heatContract)
                , uint256(amountToPay))
        );
    } else if (amountToPay < int256(0)) {
        // si c'est négatif on donne le dela au user
@>     require(heatContract.transfer(msg.sender, uint256(amountToPay *
    -1)));
```

```
// Reactor contract
// https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/
    Tokens/Reactor.sol#L33
@>   require(_amount <= guAddress.balanceOf(msg.sender), "No enough
    balance");
```

Kugle team: Fixed in commit 63307bcd092887e1f93bb07d9566cc9f94417254.

[INFO-09] Use a mapping instead of an array to cycle through authorised elements

Description:

In KugleTokenFactory, the variable `address[] authorizedContracts` is created. This variable is supposed to hold authorised addresses, allowed to transfer tokens from the token contract to users.

This could be simplified by the use of a mapping instead :

```
- address[] private authorizedContracts;
+ mapping(address => bool) public authorizedContracts;

// Remove functions `authorizeContract`, `unauthorizeContract`, `
    isAuthorizedContract`
+ function setAuthorizeContract(address _contract, bool _authorized)
    public onlyOwner {
+     authorizedContracts[_contract] = _authorized;
+ }
```

Kugle team: Fixed in commit `f6470c2552fd7ffaf2a8a1ee38d3297191095de8`.

[INFO-10] Unused variables, functions and modifiers should be removed

Description:

There are different variables, functions and modifier that are unusued or should be removed.

Deleting them allows for: - Reduced deployment and transaction costs. - Improved smart contract security. - More readable and maintainable code.

Mitigation :

Consider removing the following variables and functions : - Variable `_tokenIdSecondGen` is unused - Variable `_initialSeries` is unused - Variable `_maxSupply` should be removed - Modifier `onlyKugleOwner`is unused - Function `updateMaxSupply` should be removed - Function `resetKugleClaimDate()` should be removed - Variables `_rewardsInfos.dailyCarbonPerKugle` & `_rewardsInfos.assignedCarbonRewards` are only used by being multiplied to each other. Consider using 1 variable instead

Kugle team: Fixed in commit `d6a6465621b192ddfbf6a7919c7a703dde984127`.

[INFO-11] uint variable will always be higher than 0

There are several parts of the code that checks whether a uint variable sent in parameter is equal or superior than 0.

Because this check is always true, it should not be tested for.

Occurences : KugleFactory::addOrDeleteGuRewardTier

```
// index >= 0 is always true
require(index < _guRewardTiers.length && index >= 0);
```

KugleNFT::_mintKugle

```
// require is not needed
require(_newTokenId >= 0);
```

Mitigation:

Consider removing these `require` conditions.

Kugle team: Fixed in commit 5`b60a5fa7dcd7b8024c5e5ce6c51c96cfb5e7873`.

[INFO-12] Incorrect name, use and modification of the constant variable `_maxSupply`

Description:

The variable `_maxSupply` is designed to represent the maximum number of Genesis NFTs that users can purchase, acting as a fixed limit. However, this variable is incorrectly incremented within the `breed()` function.

Additionally, in both the `breed()` and `_mintKugle()` functions, `_maxSupply` should be used to verify whether an NFT belongs to or should be classified as part of the Genesis generation. Instead, a hardcoded value for `_maxSupply` is employed.

Furthermore, the variable name is misleading and should be renamed to `GENESIS_AMOUNT` to clearly indicate that it represents the total number of Genesis NFTs rather than a dynamic supply limit.

Recommended Mitigation:

Change the variable name by the `constant GENESIS_AMOUNT`

Delete the line that increments `_maxSupply`. https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src

Use the new `constant GENESIS_AMOUNT` in `breed()` and `_mintKugle()` instead of 2000. https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Breeding.sol#L49 https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/Breeding.sol#L55 https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/KugleNFT.sol#L75

Delete `updateMaxSupply()` as the variable is a constant and should not be changed. https://github.com/KugleCorp/kugle_contracts_foundry/blob/main/src/NFT/KugleFactory.sol#L224

Kugle team: Fixed in commit 459`fc8147d8c26d5f945e012239a7531813e748c`.

[INFO-13] Condition `require(kugleLevel > 1)` is redundant

Condition `require(kugleLevel > 1)` is redundant and could be put outside both kugles types conditions.

https://github.com/KugleCorp/kugle_contracts_foundry/blob/341a21c70e2daa8a43bdd62d352c1c157eb7cd32/src/NF
L59

```
if (_kugleId1 <= 2000) {
    require(_kugleLevel[_kugleId1] > 1 && _kugleLevel[_kugleId1] < 6);
} else {
    require(_kugleLevel[_kugleId1] > 1 && _kugleLevel[_kugleId1] < 4);
}

if (_kugleId2 <= 2000) {
    require(_kugleLevel[_kugleId2] > 1 && _kugleLevel[_kugleId2] < 6);
} else {
    require(_kugleLevel[_kugleId2] > 1 && _kugleLevel[_kugleId2] < 4);
}
```

The following implementation reduces both gas consumption and total contract size.

```
uint8 maxLevelKugle1 = condition1 <= 2000 ? 6 : 4;
uint8 maxLevelKugle2 = condition2 <= 2000 ? 6 : 4;
require(someLevel1 > 1 && someLevel1 < maxLevelKugle1);
require(someLevel2 > 1 && someLevel2 < maxLevelKugle2);
```

Kugle team: Fixed in commit `57dfc279f58a019f4aa2d45ed19e0784e7180b35` and `0259f645e22c1edaddeca2ad6e835736f442f656`.

[INFO-14] Interfaces are not inherited and contain inherited functions of other contracts

The protocol's interfaces are never inherited by their implementations contracts. Most interfaces contain function that are not implemented, but inherited from other contracts :

- `KugleFactory` should inherit `IKugleNFT`.
- `KugleBoosterSeller` should inherit `IBoosterSeller`. Some implementations of the interface are not implemented in the implementation contract.
- `IKugleToken` contains ERC20 functions, with only `transferToUser` as implemented function. ERC20 functions can be inherited from the interface directly. Another use would be to cast addresses currently typed as `IKugleToken` as `IERC20` if we want to use some of these functions.
- `IReactor` : Same as previous, with the only implemented function being `getBlockedGuByAddress` .

Kugle team: Fixed in commit `eeca9d0c627dd7a3595acb6a70f1c640e17a8f95`.