Competitive Security Assessment

KaratStaking

Aug 24th, 2023

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	KaratStaking
Platform & Language	Solidity
Codebase	• https://github.com/KaratDAO/Karat-Network-Contracts/tree/haoran_test/Staking/contracts • audit commit - 2c32956ceb5d6f2d05baf2af2b184ef967a0f24b • final commit - 98a33e97ea67174417f8c1a54830d65d06da06f3
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

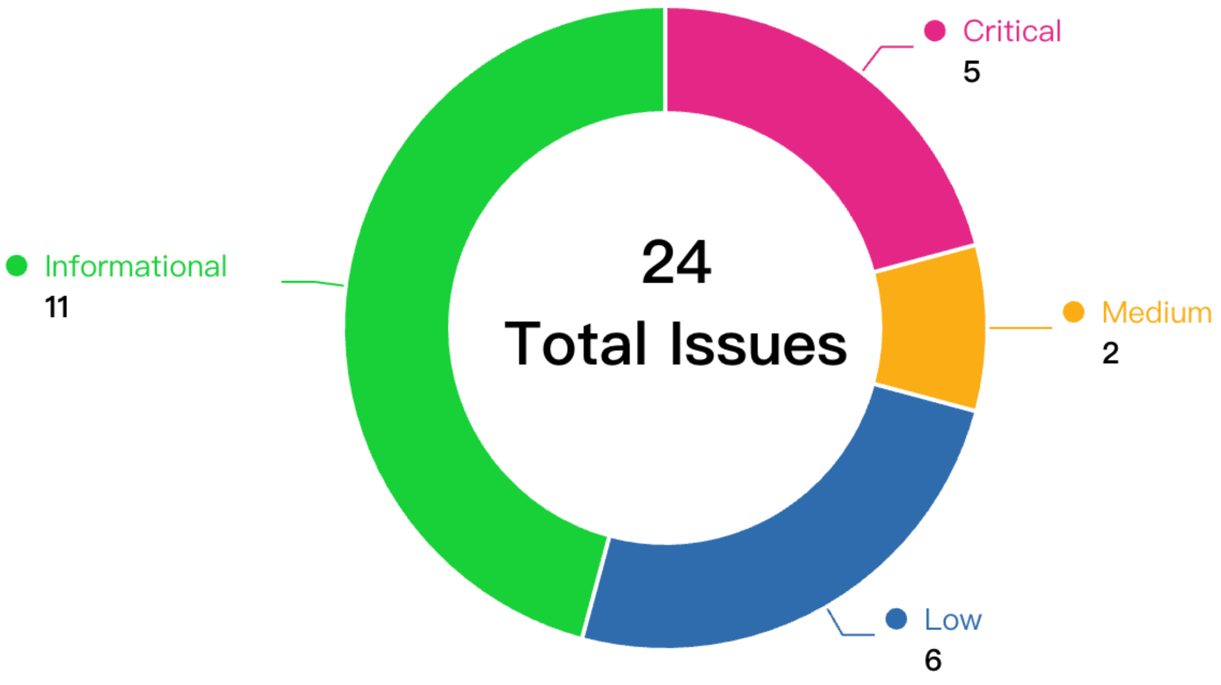
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	5	0	0	5	0	0
Medium	2	0	0	1	0	1
Low	6	0	0	4	0	2
Informational	11	0	5	4	0	2

Audit Scope

File	SHA256 Hash
./Staking_TEST/contracts/KaratStakingv2.sol	**da9ee258e8a255d392c383bc9672a0609503d993689815aee5a61e75a3ae55e4**
./Staking_TEST/contracts/KaratRewardv2.sol	**d98fd5db8e92fa6052265c20c95ac9ab6048d2282f0b290b97779cf9ba65cbfa**
./Staking_TEST/contracts/ClaimerRelayer.sol	**0df61944939c3b844c6921ec544333d5b51dfaf135ece754c63e302a3796bad0**

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
KST-1	`claimClaimer()` may Dos	Logical	Critical	Fixed	8olidity
KST-2	Possible to claim more rewards than expected	Logical	Critical	Fixed	biakia, 0x1337
KST-3	User can't claim some of their rewards	Logical	Critical	Fixed	biakia
KST-4	Malicious users can call `unstake` with any `validatorId`	Logical	Critical	Fixed	8olidity, biakia

KST-5	Anyone can call `calStaker` to clear staker's rewards	Logical	Critical	Fixed	biakia, 0x1337
KST-6	Incorrect Updating of `lastPoolUpdated` in `updatePool` may Result in Loss of Validator Benefits	Logical	Medium	Fixed	biakia, Hellobloc
KST-7	Incompatibility With Deflationary Tokens	Logical	Medium	Declined	biakia
KST-8	External calls in an un-bounded `for-loop` may result in a DOS	DOS	Low	Fixed	xfu
KST-9	Lack of check on claimed amount	Logical	Low	Fixed	biakia
KST-10	Centralization Risks	Privilege Related	Low	Declined	0x1337, 8olidity
KST-11	Logic Error in `updatePool` Function	Logical	Low	Fixed	yekong
KST-12	Missing Zero Address Check	Code Style	Low	Fixed	8olidity
KST-13	Use `disableInitializers` to prevent front-running on the initialize function	Governance Manipulation	Low	Declined	xfu, Hellobloc
KST-14	Missing events record	Logical	Informational	Acknowledged	xfu, yekong
KST-15	Unlocked Pragma Version	Language Specific	Informational	Fixed	xfu, biakia
KST-16	Redundant code	Gas Optimization	Informational	Fixed	biakia, xfu, 8olidity
KST-17	Use `calldata` instead of `memory` for function parameters	Gas Optimization	Informational	Fixed	xfu
KST-18	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	Gas Optimization	Informational	Acknowledged	xfu
KST-19	Cache state variables instead of rereading	Gas Optimization	Informational	Acknowledged	biakia, xfu
KST-20	Use indexed events for value types as they are less costly compared to non-indexed ones	Gas Optimization	Informational	Fixed	xfu
KST-21	Conformance to Solidity naming conventions	Language Specific	Informational	Acknowledged	xfu, yekong

KST-22	**Logic Error in stake Function**	**Logical**	**Informational**	**Acknowledged**	**yekong**
KST-23	**Public function that could be declared external**	**Gas Optimization**	**Informational**	**Declined**	**biakia, xfu**
KST-24	**Variables that could be declared as immutable**	**Gas Optimization**	**Informational**	**Declined**	**xfu, biakia**

KST-1: `claimClaimer()` may Dos

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	8olidity

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L98-L116

```
98:function claimClaimer(uint256 epoch) public whenNotPaused {
99:        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);
100:        require(
101:            claimerClaimedAmount[msg.sender] < totalClaimableAmount,
102:            "Already Claimed"
103:        );
104:        uint256 reward = 0;
105:        if (ifLocked) {
106:            reward =
107:                totalClaimableAmount * 10 / 100;
108:        } else {
109:            reward =
110:                totalClaimableAmount - claimerClaimedAmount[msg.sender];
111:        }
112:        claimerClaimedAmount[msg.sender] += reward;
113:
114:        _claimAndBurn(msg.sender, reward);
115:        emit RewardClaimerClaimed(msg.sender, reward, epoch);
116:    }
```

Description

8olidity : In the `claimClaimer()` function, the global value of `claimerClaimedAmount` is updated each time, but each judgment is `_staking.getClaimerRewardbyEpoch(msg.sender, epoch)`. That is to specify the reward of `epoch`, so there will be a problem. With the increase of the number of calls, since the value of `claimerClaimedAmount` is incremented each time, the situation of `claimerClaimedAmount[msg.sender] > totalClaimableAmount` will appear. And the value of `claimerClaimedAmount` cannot be reduced. The above situation is very likely to happen.

Here we write a poc to verify this situation. For portable testing, we modify the contract. The purpose of the modification is only for quick proof, not to modify the logic of the contract poc

```
//code\Staking_TEST\contracts\KaratRewardv2.sol
function claimClaimer(uint256 epoch) public whenNotPaused {
        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);
        require(
            claimerClaimedAmount[msg.sender] < totalClaimableAmount,
            "Already Claimed"
        );
        uint256 reward = 0;
        if (ifLocked) {
            // reward =
            //     totalClaimableAmount * 10 / 100;
            reward =
                totalClaimableAmount;
        } else {
            reward =
                totalClaimableAmount - claimerClaimedAmount[msg.sender];
        }
        claimerClaimedAmount[msg.sender] += reward;

        _claimAndBurn(msg.sender, reward);
        emit RewardClaimerClaimed(msg.sender, reward, epoch);
    }

// code\Staking_TEST\contracts\test\Claimer.sol
function mintClaimer(
        address to,
        uint256 validatorTokenId,
        uint256 karatScore,
        address lieutenantAddr,
        Role role
    ) public onlyRole(MINTER_ROLE) nonReentrant whenNotPaused {
        // require(balanceOf(to) < 1, "Already Have Token");
        _mintClaimer(to, validatorTokenId, karatScore, lieutenantAddr, role);
    }


// code\Staking_TEST\test\Staking.ts

describe("Reward", function () {
    it("------------------", async function () {
      const { staking, staker1, staker2, kat, relayer, reward, owner, validator1 } = await loadFixtu
re(deployStakingFixture);
```

```
  //DAY0:
        await staking.connect(staker1).stake(ethers.parseEther("100"), 1);
        const claimer1 = ethers.Wallet.createRandom().connect(ethers.provider);
        await owner.sendTransaction({
          to: claimer1.address,
          value: ethers.parseEther("100") // 100 ETH
        });
        await relayer.mintClaimer(claimer1.address, 1, 1500, ethers.ZeroAddress, 1);
        await time.increase(86400);
        const claimer2 = ethers.Wallet.createRandom().connect(ethers.provider);
        await relayer.mintClaimer(claimer2.address, 1, 1500, ethers.ZeroAddress, 1);
        await time.increase(86400);
        // const claimer3 = ethers.Wallet.createRandom().connect(ethers.provider);
        // await relayer.mintClaimer(claimer3.address, 1, 1500, ethers.ZeroAddress, 1);

        console.log("getClaimerRewardbyEpoch",await staking.getClaimerRewardbyEpoch(claimer1.address,
  0));//epoch = 0 currentEpoch = 1
        await reward.connect(claimer1).claimClaimer(0);
        console.log("claimerClaimedAmount:",await reward.claimerClaimedAmount(claimer1.address));


        await relayer.mintClaimer(claimer1.address, 1, 1500, ethers.ZeroAddress, 1);
        await time.increase(86400);

        console.log("getClaimerRewardbyEpoch",await staking.getClaimerRewardbyEpoch(claimer1.address,
  0));//epoch = 0 currentEpoch = 1
        console.log("claimerClaimedAmount:",await reward.claimerClaimedAmount(claimer1.address));
        await expect(reward.connect(claimer1).claimClaimer(0)).to.be.revertedWith("Already Claimed");
        console.log("getClaimerRewardbyEpoch",await staking.getClaimerRewardbyEpoch(claimer1.address,
  1));//epoch = 1 currentEpoch = 2

      });
```

Recommendation

8olidity : It is recommended that the claimClaimer function does not use the global claimerClaimedAmount for judgment, but each epoch corresponds to a claimerClaimedAmount

Client Response

Fixed

KST-2:Possible to claim more rewards than expected

Category	Severity	Client Response	Contributor
Logical	**Critical**	**Fixed**	**biakia, 0x1337**

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L86-L116
- code/Staking_TEST/contracts/KaratRewardv2.sol#L98-L116
- code/Staking_TEST/contracts/KaratRewardv2.sol#L78

```
ifLocked = true;

function updateIfLocked(bool newState) public onlyRole(DEFAULT_ADMIN_ROLE) {
        ifLocked = newState;
    }

    function _claimAndBurn(address to, uint256 amount) internal {
        uint256 burnAmount = (amount * 5) / 100;
        uint256 realAmount = amount - burnAmount;

        ERC20Burnable(address(_asset)).burn(burnAmount);
        SafeERC20.safeTransfer(_asset, to, realAmount);
    }

    function claimClaimer(uint256 epoch) public whenNotPaused {
        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);
        require(
            claimerClaimedAmount[msg.sender] < totalClaimableAmount,
            "Already Claimed"
        );
        uint256 reward = 0;
        if (ifLocked) {
            reward =
                totalClaimableAmount * 10 / 100;
        } else {
            reward =
                totalClaimableAmount - claimerClaimedAmount[msg.sender];
        }
        claimerClaimedAmount[msg.sender] += reward;

        _claimAndBurn(msg.sender, reward);
        emit RewardClaimerClaimed(msg.sender, reward, epoch);
    }

function claimClaimer(uint256 epoch) public whenNotPaused {
        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);
        require(
            claimerClaimedAmount[msg.sender] < totalClaimableAmount,
            "Already Claimed"
        );
    uint256 reward = 0;
    if (ifLocked) {
        reward =
            totalClaimableAmount * 10 / 100;
    } else {
        reward =
            totalClaimableAmount - claimerClaimedAmount[msg.sender];
    }
    claimerClaimedAmount[msg.sender] += reward;

    _claimAndBurn(msg.sender, reward);
    emit RewardClaimerClaimed(msg.sender, reward, epoch);
}
```

Description

biakia : In contract `RewardDistributor`, user can call `claimClaimer` function to claim their rewards:

```
function claimClaimer(uint256 epoch) public whenNotPaused {
        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);
        require(
            claimerClaimedAmount[msg.sender] < totalClaimableAmount,
            "Already Claimed"
        );
        uint256 reward = 0;
        if (ifLocked) {
            reward =
                totalClaimableAmount * 10 / 100;
        } else {
            reward =
                totalClaimableAmount - claimerClaimedAmount[msg.sender];
        }
        claimerClaimedAmount[msg.sender] += reward;

        _claimAndBurn(msg.sender, reward);
        emit RewardClaimerClaimed(msg.sender, reward, epoch);
    }
```

When `ifLocked` is true, the user will receive 1/10 rewards each time:

```
  if (ifLocked) {
              reward =
                  totalClaimableAmount * 10 / 100;
   }
```

It is possible that the user can claim more rewards than expected. Consider the `totalClaimableAmount` is 101, each time the `reward` will be `101*10/100 = 10`. After the user claims 10 times, the `claimerClaimedAmount[msg.sender]` will be 100, which is still less than `totalClaimableAmount`, so the user can claim again. At last, the user will get 110 rewards, which is greater than 101.

0x1337 : The intention of the `ifLocked` variable is that when it is set to `true` (as is the default in the constructor of the `RewardDistributor` contract), the reward that can be claimed is 10% of `totalClaimableAmount`, as shown below in line 105-107 of the `RewardDistributor` contract.

```
          if (ifLocked) {
              reward =
                  totalClaimableAmount * 10 / 100;
```

However, the claimer can simply call this function multiple times, and each time the claimer can receive 10% of total reward. If the claimer calls 10 times, he/she can receive the entirety of `totalClaimableAmount`, thus completely circumventing the intended restriction.

Recommendation

biakia : Consider the following fix:

```
function claimClaimer(uint256 epoch) public whenNotPaused {
        uint256 totalClaimableAmount = _staking.getClaimerRewardbyEpoch(msg.sender, epoch);

        uint256 reward = 0;
        if (ifLocked) {
            reward =
                totalClaimableAmount * 10 / 100;
        } else {
            reward =
                totalClaimableAmount - claimerClaimedAmount[msg.sender];
        }
        claimerClaimedAmount[msg.sender] += reward;
         require(
            claimerClaimedAmount[msg.sender] <= totalClaimableAmount,
            "Already Claimed"
        );

        _claimAndBurn(msg.sender, reward);
        emit RewardClaimerClaimed(msg.sender, reward, epoch);
    }
```

0x1337 : If the intention is to limit reward payout to 10%, then there needs to be a requirement that the function can be called only once by each claimer in each epoch.

Client Response

Fixed

KST-3:User can't claim some of their rewards

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	biakia

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L240-L267

```
240:function unstake(
241:        uint256 tokenId,
242:        uint256 validatorId
243:    ) public whenNotPaused {
244:        require(ownerOf(tokenId) == msg.sender, "not the owner");
245:        require(
246:            stakedTime[tokenId] + 86400 <= block.timestamp,
247:            "You need to Stake At Least 24 hours"
248:        );
249:        require(
250:            earned[validatorId][tokenId] == 0,
251:            "Please Claim Reward First!"
252:        );
253:
254:        _updateRewards(tokenId, validatorId);
255:        uint256 amount = tokenBalance[tokenId];
256:
257:        _burn(tokenId);
258:        delete validatorIdMapping[tokenId];
259:        poolSize[validatorId] -= 1;
260:        tokenBalance[tokenId] = 0;
261:        totalKAT -= amount;
262:        poolBalance[validatorId] -= amount;
263:        uint256 burnedAmount = (amount * 5) / 100;
264:        ERC20Burnable(address(_asset)).burn(burnedAmount);
265:        SafeERC20.safeTransfer(_asset, msg.sender, amount - burnedAmount);
266:        emit TokenUnstaked(msg.sender, validatorId, tokenId, amount);
267:    }
```

Description

biakia : In contract `StakedKaratPoolToken`, the user can call `unstake` to get back their staked tokens:

```
function unstake(
        uint256 tokenId,
        uint256 validatorId
    ) public whenNotPaused {
        require(ownerOf(tokenId) == msg.sender, "not the owner");
        require(
            stakedTime[tokenId] + 86400 <= block.timestamp,
            "You need to Stake At Least 24 hours"
        );
        require(
            earned[validatorId][tokenId] == 0,
            "Please Claim Reward First!"
        );

        _updateRewards(tokenId, validatorId);
        uint256 amount = tokenBalance[tokenId];

        _burn(tokenId);
        delete validatorIdMapping[tokenId];
        poolSize[validatorId] -= 1;
        tokenBalance[tokenId] = 0;
        totalKAT -= amount;
        poolBalance[validatorId] -= amount;
        uint256 burnedAmount = (amount * 5) / 100;
        ERC20Burnable(address(_asset)).burn(burnedAmount);
        SafeERC20.safeTransfer(_asset, msg.sender, amount - burnedAmount);
        emit TokenUnstaked(msg.sender, validatorId, tokenId, amount);
    }
```

It will call `_updateRewards` to update user's latest rewards:

```
function _updateRewards(
        uint256 stakerTokenId,
        uint256 validatorId
    ) private {
        earned[validatorId][stakerTokenId] += _calculateRewards(
            stakerTokenId,
            validatorId
        );
        rewardIndexOf[validatorId][stakerTokenId] = rewardIndex[validatorId];
    }
```

and then burn the NFT:

```
_burn(tokenId);
```

If the user wants to claim the rewards, they should call `claimStaker` function in contract `RewardDistributor`:

```
function claimStaker(
        uint256 stakerTokenId,
        uint256 validatorId
    ) public whenNotPaused {
        require(
            _staking.ownerOf(stakerTokenId) == msg.sender,
            "Not Owner of this Token"
        );
        uint256 reward = _staking.calStaker(stakerTokenId, validatorId);
        require(reward > 0, "No Reward to Claim");

        _claimAndBurn(msg.sender, reward);

        emit RewardStakerClaimed(msg.sender, reward);
    }
```

This function will check whether the `msg.sender` is the owner of the NFT. However, since the NFT has already been destroyed in `unstake` function, this call will revert. At last, the user's rewards will be frozen forever.

Recommendation

biakia : Consider redesigning the logic of reward distribution.

Client Response

Fixed

KST-4:Malicious users can call `unstake` with any `validator Id`

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	8olidity, biakia

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L240-L267
- code/Staking_TEST/contracts/KaratStakingv2.sol#L241-L242

```
function unstake(
        uint256 tokenId,
        uint256 validatorId
    ) public whenNotPaused {
        require(ownerOf(tokenId) == msg.sender, "not the owner");
        require(
            stakedTime[tokenId] + 86400 <= block.timestamp,
            "You need to Stake At Least 24 hours"
        );
        require(
            earned[validatorId][tokenId] == 0,
            "Please Claim Reward First!"
        );

        _updateRewards(tokenId, validatorId);
        uint256 amount = tokenBalance[tokenId];

        _burn(tokenId);
        delete validatorIdMapping[tokenId];
        poolSize[validatorId] -= 1;
        tokenBalance[tokenId] = 0;
        totalKAT -= amount;
        poolBalance[validatorId] -= amount;
        uint256 burnedAmount = (amount * 5) / 100;
        ERC20Burnable(address(_asset)).burn(burnedAmount);
        SafeERC20.safeTransfer(_asset, msg.sender, amount - burnedAmount);
        emit TokenUnstaked(msg.sender, validatorId, tokenId, amount);
    }

uint256 tokenId,
        uint256 validatorId
```

Description

8olidity : When `stake()`, it will ask to enter `validatorId`, and then cast a `tokenid` to `msg.sender`. But when `unstake()`, the association between `tokenid` and `validatorId` is not checked. The user can output the wrong `validatorId`, reduce the balance of the corresponding `poolBalance` and the value of `poolSize`

biakia : In the `unstake` function, there is no check for `validatorId`, which can lead to potential attack. Consider the following attack flow:

1. Alice stakes tokens with `validatorId` 1

2. Bob stakes tokens with `validatorId` 2
3. In this case, the value of `poolSize[1]` and `poolSize[2]` are both 1.
4. Bob is an attacker and he calls `unstake` function with `validatorId=1`, at this time, `poolSize[1]` will be 0 due to the following code:

```
poolSize[validatorId] -= 1;
```

5. Alice is a normal user and she calls `unstake` function with `validatorId=1`. Since the `poolSize[1]` is 0 now, the call will revert due to an underflow error.

Further more, the rewards calculated in `_updateRewards` will be inaccurate and the number of `poolBalance[validatorId]` will also be inaccurate due to this attack.

Recommendation

8olidity : It is recommended to check the association between `tokenid` and `validatorId` when performing `unstake()`
biakia : Consider adding a check for `validatorId`:

```
function unstake(
        uint256 tokenId,
        uint256 validatorId
    ) public whenNotPaused {
        require(ownerOf(tokenId) == msg.sender, "not the owner");
        require(validatorIdMapping[tokenId] == validatorId,"invalid validatorId");
        ...
        ...
    }
```

Client Response

Fixed

KST-5:Anyone can call `calStaker` to clear staker's rewards

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	biakia, 0x1337

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L371-L381
- code/Staking_TEST/contracts/KaratRewardv2.sol#L181-L195

```
181:function claimStaker(
182:        uint256 stakerTokenId,
183:        uint256 validatorId
184:    ) public whenNotPaused {
185:        require(
186:            _staking.ownerOf(stakerTokenId) == msg.sender,
187:            "Not Owner of this Token"
188:        );
189:        uint256 reward = _staking.calStaker(stakerTokenId, validatorId);
190:        require(reward > 0, "No Reward to Claim");
191:
192:        _claimAndBurn(msg.sender, reward);
193:
194:        emit RewardStakerClaimed(msg.sender, reward);
195:    }

371:function calStaker(
372:        uint256 stakerTokenId,
373:        uint256 validatorId
374:    ) public returns (uint256) {
375:        _updateRewards(stakerTokenId, validatorId);
376:        uint256 reward = earned[validatorId][stakerTokenId] / INDEX_MULTIPLIER;
377:        if (reward > 0) {
378:            earned[validatorId][stakerTokenId] = 0;
379:        }
380:        return reward;
381:    }
```

Description

biakia : In contract `StakedKaratPoolToken`, the function `calStaker` does not have privilege checks:

```
function calStaker(
        uint256 stakerTokenId,
        uint256 validatorId
    ) public returns (uint256) {
        _updateRewards(stakerTokenId, validatorId);
        uint256 reward = earned[validatorId][stakerTokenId] / INDEX_MULTIPLIER;
        if (reward > 0) {
            earned[validatorId][stakerTokenId] = 0;
        }
        return reward;
    }
```

If the `earned[validatorId][stakerTokenId]` is greater than `INDEX_MULTIPLIER`, then the `reward` will be greater than 0. In this case, the `earned[validatorId][stakerTokenId]` will be set as 0, which means the staker will lose all rewards. Malicious users can call this function to clear all stakers' rewards.
0x1337 : The `claimStaker()` function in the `RewardDistributor` contract is used by stakers to claim their rewards. The function contains a check that the `msg.sender` is the owner of the particular `stakerTokenId`. The function calls the `calStaker()` function of the `_staking` contract to derive the amount of reward that the staker is eligible for. The critical vulnerability here is that in the `_staking` (`StakedKaratPoolToken`) contract, the `calStaker()` function is unprotected and can be called by anyone, and in this function, `earned[validatorId][stakerTokenId]` is set to 0 if the reward amount is greater than 0. This means that an attacker could call this `calStaker()` function directly, and for any staker with a reward amount greater than 0, the attacker could set its `earned[validatorId][stakerTokenId]` to 0, such that when the staker tries to claim its reward, the reward will be 0 because line 376 will return 0. The value at risk is all reward for all stakers.

Recommendation

biakia : Consider restricting the calling privileges of this function, for example, making sure that this function is only allowed to be called by the contract `RewardDistributor`.
0x1337 : The `calStaker()` function needs to be protected (i.e. add access control) so that it cannot be called by anyone. If there's a need for a view function, then create a separate view function that does not change the value of the `earned` mapping.

Client Response

Fixed

KST-6:Incorrect Updating of `lastPoolUpdated` in `updatePool` may Result in Loss of Validator Benefits

Category	Severity	Client Response	Contributor
Logical	**Medium**	**Fixed**	**biakia, Hellobloc**

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L282-L310
- code/Staking_TEST/contracts/KaratStakingv2.sol#L282-L303

```
function updatePool(
        uint256 validatorId,
        uint256 epoch
    ) public onlyRole(AUTHORIZED_CALLER) {
        require(epoch < currentEpoch, "This Epoch is not Valid");
        require(currentEpoch >= 1, "Must Calculate After first Day");
        require(!isValidatorClaimed[validatorId][epoch], "Rewards Calculated");

        if (validatorReward[validatorId][epoch] != 0) {
            isValidatorClaimed[validatorId][epoch] = true;
            validatorEarns[validatorId] +=
                (validatorReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) /
                totalKaratReward[epoch];

            //Update Pool Reward For Stakers
            _updateRewardIndex(
                validatorId,
                ((stakerReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) / totalKaratReward[epoch])
            );
        }

        require(
            epoch >= lastPoolUpdated[validatorId],
            "Cannot Update Before Days"
        );
        lastPoolUpdated[validatorId] = epoch + 1;
    }
```

```
function updatePool(
        uint256 validatorId,
        uint256 epoch
    ) public onlyRole(AUTHORIZED_CALLER) {
        require(epoch < currentEpoch, "This Epoch is not Valid");
        require(currentEpoch >= 1, "Must Calculate After first Day");
        require(!isValidatorClaimed[validatorId][epoch], "Rewards Calculated");

        if (validatorReward[validatorId][epoch] != 0) {
            isValidatorClaimed[validatorId][epoch] = true;
            validatorEarns[validatorId] +=
                (validatorReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) /
                totalKaratReward[epoch];

            //Update Pool Reward For Stakers
            _updateRewardIndex(
                validatorId,
                ((stakerReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) / totalKaratReward[epoch])
            );
        }
```

Description

biakia : The function `updatePool` is used to update rewards of each epoch for validators and stakers:

```
function updatePool(
        uint256 validatorId,
        uint256 epoch
    ) public onlyRole(AUTHORIZED_CALLER) {
        require(epoch < currentEpoch, "This Epoch is not Valid");
        require(currentEpoch >= 1, "Must Calculate After first Day");
        require(!isValidatorClaimed[validatorId][epoch], "Rewards Calculated");

        if (validatorReward[validatorId][epoch] != 0) {
            isValidatorClaimed[validatorId][epoch] = true;
            validatorEarns[validatorId] +=
                (validatorReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) /
                totalKaratReward[epoch];

            //Update Pool Reward For Stakers
            _updateRewardIndex(
                validatorId,
                ((stakerReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) / totalKaratReward[epoch])
            );
        }

        require(
            epoch >= lastPoolUpdated[validatorId],
            "Cannot Update Before Days"
        );
        lastPoolUpdated[validatorId] = epoch + 1;
    }
```

Consider the `lastPoolUpdated[validatorId]` now is 1 and the input param `epoch` is 5. After calling `updatePool`, the rewards of the epoch 5 will be updated and the `lastPoolUpdated[validatorId]` will be set as 6. After that, the rewards of the epoch 2,3,4 will never be updated because the following check will never pass:

```
require(
            epoch >= lastPoolUpdated[validatorId],
            "Cannot Update Before Days"
);
```

That means these rewards will be lost.

Hellobloc : The `tryUpdateSnapshot` as well as `tryUpdatePool` provide methods for updating the `Pool` as well as the `Snapshot`, which will be called in `mintClaimer`.

However, they may be limited by gas, causing the call to revert. Specifically, the "tryUpdateSnapshot" and "tryUpdatePool" methods loop based on "currentEpoch", as in the following code:

```
function tryUpdatePool(
        uint256 validatorId
    ) public onlyRole(AUTHORIZED_CALLER) {
        uint256 lastUpdate = lastPoolUpdated[validatorId];
        if (currentEpoch > 0 && lastUpdate <= currentEpoch - 1) {
            for (uint i = lastUpdate; i < currentEpoch; i++) {
                updatePool(validatorId, i);
            }
        }
    }
```

When `currentEpoch` is not updated frequently enough by `tryUpdateSnapshot()`, it can lead to too many loops and cause the transaction to reach the gas limit triggering a revert. So the contract provides the `Snapshot` and `update Pool` public methods for single epoch updates to protect against dos risk due to gas limit.

However, the `updatePool` method has the risk of centralization, where a malicious `AUTHORIZED_CALLER` can update the `lastPoolUpdatedEpoch` of the validator non-continuously, which will result in the `validatorEarns` not being properly updated and thus the benefits will be damaged.

```
function updatePool(
        uint256 validatorId,
        uint256 epoch
    ) public onlyRole(AUTHORIZED_CALLER) {
        require(epoch < currentEpoch, "This Epoch is not Valid");
        require(currentEpoch >= 1, "Must Calculate After first Day");
        require(!isValidatorClaimed[validatorId][epoch], "Rewards Calculated");
        if (validatorReward[validatorId][epoch] != 0) {
            //reward update
            ...
        }
        require(
            epoch >= lastPoolUpdated[validatorId],
            "Cannot Update Before Days"
        );
        lastPoolUpdated[validatorId] = epoch + 1;
    }
```

For example, in the above code, even if the `lastPoolUpdated` of the current validator is only `currentEpoch-10`, `AUTHORIZED_CALLER` can still update only the reward of `currentEpoch-1` and ignore the reward of `currentEpoch-10` to `currentEpoch-2`.

Recommendation

biakia : Consider adding a check to make sure only the next epoch can be updated:

```
function updatePool(
        uint256 validatorId,
        uint256 epoch
    ) public onlyRole(AUTHORIZED_CALLER) {
        require(epoch < currentEpoch, "This Epoch is not Valid");
        require(currentEpoch >= 1, "Must Calculate After first Day");
        require(epoch == lastPoolUpdated[validatorId]+1,"invalid epoch");
        ....
```

Hellobloc : We recommend updating the `updatePool` method so that it only allows sequential updates to `lastPoolUpdated`.

Client Response

Fixed

KST-7:Incompatibility With Deflationary Tokens

Category	Severity	Client Response	Contributor
Logical	Medium	Declined	biakia

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L219-L238

```
219:function stake(
220:        uint256 amount,
221:        uint256 validatorId
222:    ) public nonReentrant whenNotPaused {
223:        require(validatorId != 0, "Pool 0 is open for staking");
224:        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
225:        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
226:        safeMint(msg.sender, validatorId, amount);
227:    }
228:
229:    function stakeTo(
230:        address recipient,
231:        uint256 amount,
232:        uint256 validatorId
233:    ) public nonReentrant whenNotPaused {
234:        require(validatorId != 0, "Pool 0 is open for staking");
235:        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
236:        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
237:        safeMint(recipient, validatorId, amount);
238:    }
```

Description

biakia : In contract `StakedKaratPoolToken`, the function `stake` and `stakeTo` will transfer reward tokens to the contract:

```
  function stake(
          uint256 amount,
          uint256 validatorId
      ) public nonReentrant whenNotPaused {
          require(validatorId != 0, "Pool 0 is open for staking");
          require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
          SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
          safeMint(msg.sender, validatorId, amount);
      }

      function stakeTo(
          address recipient,
          uint256 amount,
          uint256 validatorId
      ) public nonReentrant whenNotPaused {
          require(validatorId != 0, "Pool 0 is open for staking");
          require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
          SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
          safeMint(recipient, validatorId, amount);
      }
```

If the `_asset` is a deflationary token, the input param `amount` may not be equal to the received amount due to the charged transaction fee. As a result, an inconsistency in the amount will occur and the contract will not have enough tokens to pay back to users when they call `unstake`.

Recommendation

biakia : Consider recording the actual rewards in function `stake` and `stakeTo`:

```
function stake(
        uint256 amount,
        uint256 validatorId
    ) public nonReentrant whenNotPaused {
        require(validatorId != 0, "Pool 0 is open for staking");
        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
        uint256 beforeBalance = _asset.balanceOf(address(this));
        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
        amount = _asset.balanceOf(address(this)) - beforeBalance;
        safeMint(msg.sender, validatorId, amount);
    }

    function stakeTo(
        address recipient,
        uint256 amount,
        uint256 validatorId
    ) public nonReentrant whenNotPaused {
        require(validatorId != 0, "Pool 0 is open for staking");
        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
        uint256 beforeBalance = _asset.balanceOf(address(this));
        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
        amount = _asset.balanceOf(address(this)) - beforeBalance;
        safeMint(recipient, validatorId, amount);
    }
```

Client Response

Declined

KST-8:External calls in an un-bounded `for-loop` may result in a DOS

Category	Severity	Client Response	Contributor
DOS	Low	Fixed	xfu

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L156-L159
- code/Staking_TEST/contracts/KaratRewardv2.sol#L170-L173

```
156:totalRewards += _staking.getLieutenantRewardbyEpoch(
157:                    msg.sender,
158:                    epochs[i]
159:                );

170:totalRewards += _staking.getLieutenantRewardbyEpoch(
171:                    msg.sender,
172:                    i
173:                );
```

Description

xfu : The use of external calls in nested loops and subsequent loops, which iterate over lists that could have been provided by callers, may result in an out-of-gas failure during execution.

There are `2` instances of this issue:

- totalRewards += _staking.getLieutenantRewardbyEpoch(msg.sender,epochs[i]) external calls in loop may result in DOS.
- totalRewards += _staking.getLieutenantRewardbyEpoch(msg.sender,i_scope_0) external calls in loop may result in DOS.

Recommendation

xfu : It is recommended to set the max length to which a for loop can iterate. If possible, use pull over push strategy for external calls.

Client Response

Fixed

KST-9:Lack of check on claimed amount

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	biakia

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L118-L131

```
function claimValidator(uint256 validatorId) public whenNotPaused {
        require(
            _validator.ownerOf(validatorId) == msg.sender,
            "Not the Token Owner"
        );

        uint256 totalReward = _staking.getClaimValidatorReward(validatorId);
        uint256 unclaimed = totalReward - validatorRewardsClaimed[validatorId];
        validatorRewardsClaimed[validatorId] = totalReward;

        _claimAndBurn(msg.sender, unclaimed);

        emit RewardValidatorClaimed(validatorId, unclaimed);
    }
```

Description

biakia : The function `claimValidator` will call `_claimAndBurn` even though the `unclaimed` is 0:

```
function claimValidator(uint256 validatorId) public whenNotPaused {
        require(
            _validator.ownerOf(validatorId) == msg.sender,
            "Not the Token Owner"
        );

        uint256 totalReward = _staking.getClaimValidatorReward(validatorId);
        uint256 unclaimed = totalReward - validatorRewardsClaimed[validatorId];
        validatorRewardsClaimed[validatorId] = totalReward;

        _claimAndBurn(msg.sender, unclaimed);

        emit RewardValidatorClaimed(validatorId, unclaimed);
    }
```

If `unclaimed` is 0, the user calling this function will waste gas and get nothing.

Recommendation

biakia : Consider adding a check on `unclaimed`:

```
function claimValidator(uint256 validatorId) public whenNotPaused {
        require(
            _validator.ownerOf(validatorId) == msg.sender,
            "Not the Token Owner"
        );

        uint256 totalReward = _staking.getClaimValidatorReward(validatorId);
        uint256 unclaimed = totalReward - validatorRewardsClaimed[validatorId];
        require(unclaimed>0,"unclaimed is 0");
        validatorRewardsClaimed[validatorId] = totalReward;

        _claimAndBurn(msg.sender, unclaimed);

        emit RewardValidatorClaimed(validatorId, unclaimed);
    }
```

Client Response

Fixed

KST-10:Centralization Risks

Category	Severity	Client Response	Contributor
Privilege Related	**Low**	**Declined**	**0x1337, 8olidity**

Code Reference

- code/Staking_TEST/contracts/ClaimerRelayer.sol#L89-L100
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L102-L128
- code/Staking_TEST/contracts/KaratRewardv2.sol#L86-L88
- code/Staking_TEST/contracts/KaratRewardv2.sol#L197-L216
- code/Staking_TEST/contracts/KaratStakingv2.sol#L161-L192
- code/Staking_TEST/contracts/KaratStakingv2.sol#L427-L441
- code/Staking_TEST/contracts/KaratRewardv2.sol#L197-L202

```
function updateIfLocked(bool newState) public onlyRole(DEFAULT_ADMIN_ROLE) {
        ifLocked = newState;
    }

function changeReward(
        uint256 newReward
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        rewardAmount = newReward;
    }

    function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_katToken, to, amount);
    }

function mintClaimer(
        address to,
        uint256 validatorTokenId,
        uint256 karatScore,
        address lieutenantAddr,
        Role role
    ) public onlyRole(AUTHORIZED_CALLER) {
        _claimer.mintClaimer(
            to,
            validatorTokenId,
            karatScore,
            lieutenantAddr,
            role
        );

        bool result = _staking.tryUpdateSnapshot(to);
        if (result) {
            SafeERC20.safeTransfer(_katToken, to, rewardAmount);
        }
        _staking.tryUpdatePool(validatorTokenId);
        _staking.updateClaimerReward(
            validatorTokenId,
            to,
            lieutenantAddr,
            karatScore
        );
    }

function tryUpdateSnapshot(
        address claimer
    ) public onlyRole(AUTHORIZED_CALLER) returns (bool) {
        uint256 startEpochTime = (currentEpoch + 1) * 86400 + firstDAYUnix;
        if (
            block.timestamp >= startEpochTime &&
            ifEverydayRewardClaimed[currentEpoch + 1] == address(0)
        ) {
            uint256 days_gapped = (block.timestamp - startEpochTime) /
                86400 +
                1;
            for (uint i = 0; i < days_gapped; i++) {
                ifEverydayRewardClaimed[currentEpoch + 1] = claimer;
                _snapshot();
            }
            return true;
        } else {
            return false;
        }
    }

    function tryUpdatePool(
        uint256 validatorId
    ) public onlyRole(AUTHORIZED_CALLER) {
        uint256 lastUpdate = lastPoolUpdated[validatorId];

        if (currentEpoch > 0 && lastUpdate <= currentEpoch - 1) {
            for (uint i = lastUpdate; i < currentEpoch; i++) {
                updatePool(validatorId, i);
            }
        }
    }

function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_asset, to, amount);
    }

    // Function to pause all token minting, accessible only by admin
    function pause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    // Function to unpause all token minting, accessible only by admin
    function unpause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }

    function _authorizeUpgrade(
        address
    ) internal override onlyRole(DEFAULT_ADMIN_ROLE) {}

function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_asset, to, amount);
    }

function setMinimumToStake(
        uint256 minimumToStake_
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        minimumToStake = minimumToStake_;
    }

    // Function to pause all token minting, accessible only by admin
    function pause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    // Function to unpause all token minting, accessible only by admin
    function unpause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

Description

0x1337 : The privileged roles have significant privileges in the code base. For example, privileged roles can pause/unpause contracts, upgrade the contract, withdraw tokens, etc.. In addition to what the privileged roles can do, what the privileged roles choose not to do can also have a significant impact on the project. For example, it can choose not to call functions such as `mintClaimer()` of the `ClaimerRelayer` contract, in which case reward tokens won't be

available for stakers. If the privileged role is compromised, the entire project is at risk.
8olidity : The administrator can use the `withdrawToken()` function to transfer the `asset` in the reward contract, so that the user cannot get the `reward`

```
    function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_asset, to, amount);
    }
```

Recommendation

0x1337 : Consider using multisig + timelock for privileged roles, and carefully manage the private keys.
8olidity : It is recommended to limit the amount parameter in the withdrawToken function, and it is not allowed to transfer all the funds in the contract. If it is for an emergency, you can rewrite an emergency rescue function

Client Response

Declined

KST-11:Logic Error in `updatePool` Function

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	yekong

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L305-L309

```
305:require(
306:            epoch >= lastPoolUpdated[validatorId],
307:            "Cannot Update Before Days"
308:        );
309:        lastPoolUpdated[validatorId] = epoch + 1;
```

Description

yekong : There is a logic error in the updatePool function. The function performs all its update operations and only then does it check if the epoch is greater than or equal to lastPoolUpdated[validatorId]. This is a logic error as the function should fail immediately if the epoch is not greater than or equal to lastPoolUpdated[validatorId] and not perform any update operations.

Recommendation

yekong : Move the require(epoch >= lastPoolUpdated[validatorId], "Cannot Update Before Days"); condition check to the beginning of the function, ensuring that all preconditions are met before performing any update operations.

Client Response

Fixed

KST-12:Missing Zero Address Check

Category	Severity	Client Response	Contributor
Code Style	Low	Fixed	8olidity

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L386-L391
- code/Staking_TEST/contracts/KaratStakingv2.sol#L230-L237
- code/Staking_TEST/contracts/KaratStakingv2.sol#L162
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L110

```
110:to,

162:address claimer

230:address recipient,
231:        uint256 amount,
232:        uint256 validatorId
233:    ) public nonReentrant whenNotPaused {
234:        require(validatorId != 0, "Pool 0 is open for staking");
235:        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
236:        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
237:        safeMint(recipient, validatorId, amount);

386:address claimerAddress,
387:        address lieutenantAddr,
388:        uint256 karatScore
389:    ) public onlyRole(AUTHORIZED_CALLER) {
390:        uint256 claimerR = (karatScore * 2) / 3;
391:        claimerReward[claimerAddress][currentEpoch] = claimerR;
```

Description

8olidity : Many functions in the contract did not judge the incoming address to determine whether it is an address(0) address. If some key addresses are set to address(0), it may cause damage to the protocol

Recommendation

8olidity : It is recommended to judge whether the address is address(0)

Client Response

Fixed

KST-13:Use `disableInitializers` to prevent front-running on the initialize function

Category	Severity	Client Response	Contributor
Governance Manipulation	**Low**	**Declined**	**xfu, Hellobloc**

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L23-L33
- code/Staking_TEST/contracts/KaratRewardv2.sol#L43-L48
- code/Staking_TEST/contracts/KaratRewardv2.sol#L43
- code/Staking_TEST/contracts/KaratStakingv2.sol#L23

```
23:contract StakedKaratPoolToken is

23:contract StakedKaratPoolToken is
24:    Initializable,
25:    ERC721Upgradeable,
26:    ERC721EnumerableUpgradeable,
27:    ERC721URIStorageUpgradeable,
28:    ERC721BurnableUpgradeable,
29:    AccessControlUpgradeable,
30:    UUPSUpgradeable,
31:    ReentrancyGuardUpgradeable,
32:    PausableUpgradeable
33:{

43:contract RewardDistributor is

43:contract RewardDistributor is
44:    Initializable,
45:    AccessControlUpgradeable,
46:    UUPSUpgradeable,
47:    PausableUpgradeable
48:{
```

Description

xfu : The implementation contracts behind a proxy can be initialized by any address. This is not a security problem in the sense that it impacts the system directly, as the attacker will not be able to cause any contract to self-destruct or modify any values in the proxy contracts. However, taking ownership of implementation contracts can open other attack vectors, like social engineering or phishing attacks.

More detail see this OpenZeppelin docs and this.

There are `2` instances of this issue:

- `RewardDistributor` in `KaratRewardv2.sol` is an upgradeable contract that does not protect its initialize functions:

```
    function initialize(
        IERC20 katToken_,
        IStakingContract staking_,
        IValidatorContract validator_
    ) public initializer {
```

- `StakedKaratPoolToken` in `KaratStakingv2.sol` is an upgradeable contract that does not protect its initialize functions:

```
    function initialize(
        IERC20 asset_,
        uint256 firstdayUnixTime
    ) public initializer {
```

Hellobloc : The Karat project follows zksync's recommendations very well, e.g. using upgradeable contracts to implement its project. But please don't leave an implementation contract uninitialized for the sake of best practices. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy.

To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed.

Recommendation

xfu : Use disableInitializers to prevent front-running on the initialize func-tion, as it would make you deploy the smart contract again if someone initializes it before you.

```
    constructor(){
        _disableInitializers();
    }
```

Hellobloc : We recommend adding the following constructor to the contract to prevent malicious initialization of the implementation contract.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Client Response

Declined

KST-14:Missing events record

Category	Severity	Client Response	Contributor
Logical	**Informational**	**Acknowledged**	**xfu, yekong**

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L86-L88
- code/Staking_TEST/contracts/KaratRewardv2.sol#L197-L202
- code/Staking_TEST/contracts/KaratStakingv2.sol#L427-L431
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L89-L93
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L95-L100
- code/Staking_TEST/contracts/KaratRewardv2.sol#L130

```
function updateIfLocked(bool newState) public onlyRole(DEFAULT_ADMIN_ROLE) {
        ifLocked = newState;
    }

function changeReward(
        uint256 newReward
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        rewardAmount = newReward;
    }

function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_katToken, to, amount);
    }

emit RewardValidatorClaimed(validatorId, unclaimed);

function withdrawToken(
        address to,
        uint256 amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        SafeERC20.safeTransfer(_asset, to, amount);
    }

function setMinimumToStake(
        uint256 minimumToStake_
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        minimumToStake = minimumToStake_;
    }
```

Description

xfu : Setter-functions must emit events

There is `1` instance of this issue:

- Setter function StakedKaratPoolToken.setMinimumToStake(uint256) does not emit an event
  xfu : When an action is triggered based on a user's action, not being able to filter based on who triggered the action makes event processing a lot more cumbersome. Including the `msg.sender` the events of these types of action will make events much more useful to end users.

There is `1` instance of this issue:

- RewardValidatorClaimed(validatorId,unclaimed) should add `msg.sender` to event.
  yekong : Events are crucial in smart contracts, as they facilitate efficient communication between the contract and its users, as well as with other contracts. They are instrumental in tracking contract state changes and making the contract more transparent. We identified several functions in the contract where significant actions were taken (e.g., state changes, token transfers, etc.) without corresponding event emissions. This could hinder monitoring and verification of these operations on-chain, making it harder for users or external systems to react to these actions.

Recommendation

xfu : Emit events in setter functions.

For example:

```
    event MinimumToStakeChanged(uint256);

    function setMinimumToStake(
      uint256 minimumToStake_
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
      minimumToStake = minimumToStake_;
      emit MinimumToStakeChanged(minimumToStake_);
    }
```

xfu : Adding `msg.sender` to event.

For example:

```
      emit RewardValidatorClaimed(msg.sender, validatorId, unclaimed);
```

yekong : We recommend emitting appropriate events in all functions where significant actions are taken. Specifically, for any state changes, token transfers, access control changes, or other notable operations, ensure that an event is emitted with all relevant details. This will not only improve transparency and traceability of the contract's actions but also enable efficient interactions with users or external systems.

Client Response

Acknowledged

KST-15:Unlocked Pragma Version

Category	Severity	Client Response	Contributor
Language Specific	**Informational**	**Fixed**	**xfu, biakia**

Code Reference

- code/Staking_TEST/contracts/KaratRewardv2.sol#L2
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L2
- code/Staking_TEST/contracts/KaratStakingv2.sol#L2

```
2:pragma solidity ^0.8.18;

2:pragma solidity ^0.8.18;

2:pragma solidity ^0.8.18;
```

Description

xfu : Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

More detail see SWC-103.

There are `3` instances of this issue:

- Should lock the pragma version instead of floating pragma: ^0.8.18.
- Should lock the pragma version instead of floating pragma: ^0.8.18.
- Should lock the pragma version instead of floating pragma: ^0.8.18.

  xfu : Solidity `0.8.21` has many optimization with compiler and bugfixes, please upgrade Solidity to the latest version(`0.8.21`) for gas reduction and improved security.

There are `3` instances of this issue:

- pragma solidity version ^0.8.18 should upgrade to the latest version: 0.8.21
- pragma solidity version ^0.8.18 should upgrade to the latest version: 0.8.21
- pragma solidity version ^0.8.18 should upgrade to the latest version: 0.8.21

  biakia : Solidity files in packages have a pragma version `^0.8.18`. The caret `(^)` points to unlocked pragma, meaning the compiler will use the specified version or above.

Recommendation

xfu : Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

xfu : Upgrade solidity version to the latest version: 0.8.21
biakia : It's good practice to use specific solidity versions to know compiler bug fixes and optimisations were enabled at the time of compiling the contracts.

Client Response

Fixed

KST-16:Redundant code

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	biakia, xfu, 8olidity

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L16
- code/Staking_TEST/contracts/KaratRewardv2.sol#L5
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L70
- code/Staking_TEST/contracts/KaratStakingv2.sol#L97-L100

```
5:import "@openzeppelin/contracts-upgradeable/utils/cryptography/ECDSAUpgradeable.sol";

16:import "@openzeppelin/contracts-upgradeable/utils/StringsUpgradeable.sol"; // Utility to convert
to string types

70:mapping(uint256 => address) ifEverydayRewardClaimed;

97:uint256 private constant MULTIPLIER = 1e18;
98:    uint256 private constant INDEX_MULTIPLIER = 1e18;
99:
100:    uint256 public constant startingReward = 5000000 * MULTIPLIER;
```

Description

biakia : In the contract `ClaimerRelayer`, the variable `ifEverydayRewardClaimed` is never used.
biakia : The contract `RewardDistributor` includes the following unnecessary imports:

```
import "@openzeppelin/contracts-upgradeable/utils/cryptography/ECDSAUpgradeable.sol";
```

The contract `StakedKaratPoolToken` includes the following unnecessary imports:

```
import "@openzeppelin/contracts-upgradeable/utils/StringsUpgradeable.sol";
```

xfu : Saves a storage slot. If the variable is assigned a non-zero value, saves Gsset (20000 gas). If it's assigned a zero value, saves Gsreset (2900 gas). If the variable remains unassigned, there is no gas savings unless the variable is public, in which case the compiler-generated non-payable getter deployment cost is saved. If the state variable is overriding an interface's public function, mark the variable as constant or immutable so that it does not use a storage slot
There is `1` instance of this issue:

- ClaimerRelayer.ifEverydayRewardClaimed is never used.
  8olidity : MULTIPLIER and INDEX_MULTIPLIER can delete MULTIPLIER, MULTIPLIER is only used once, and the value is the same as INDEX_MULTIPLIER, it can be used instead

```
    uint256 private constant MULTIPLIER = 1e18;
    uint256 private constant INDEX_MULTIPLIER = 1e18;
```

Recommendation

biakia : If these variables are not intended to be used, it is recommended to remove them to save gas.
biakia : Consider removing the import statement to save on deployment gas costs.
xfu : Remove or replace the unused state variables
8olidity :

```
-     uint256 private constant MULTIPLIER = 1e18;
      uint256 private constant INDEX_MULTIPLIER = 1e18;

    uint256 public constant startingReward = 5000000 * INDEX_MULTIPLIER ;
```

Client Response

Fixed

KST-17:Use `calldata` instead of `memory` for function parameters

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	xfu

Code Reference

- code/Staking_TEST/contracts/ClaimerRelayer.sol#L130-L158

```
function mintClaimerWithSig(
        address to,
        uint256 validatorTokenId,
        uint256 karatScore,
        address lieutenantAddr,
        Role role,
        bytes memory signature
    ) public {
        _claimer.mintClaimerwithSig(
            to,
            validatorTokenId,
            karatScore,
            lieutenantAddr,
            role,
            signature
        );

        bool result = _staking.tryUpdateSnapshot(to);
        if (result) {
            SafeERC20.safeTransfer(_katToken, to, rewardAmount);
        }
        _staking.tryUpdatePool(validatorTokenId);
        _staking.updateClaimerReward(
            validatorTokenId,
            to,
            lieutenantAddr,
            karatScore
        );
    }
```

Description

xfu : On external functions, when using the `memory` keyword with a function argument, what's happening is a `memory` acts as an intermediate.

When the function gets called externally, the array values are kept in `calldata` and copied to memory during ABI decoding (using the opcode `calldataload` and `mstore`). And during the for loop, the values in the array are accessed in memory using a `mload`. That is inefficient. Reading directly from `calldata` using `calldataload` instead of going via `memory` saves the gas from the intermediate memory operations that carry the values.

More detail see this

There is `1` instance of this issue:

- ClaimerRelayer.mintClaimerWithSig(address,uint256,uint256,address,Role,bytes) read-only `memory` parameters below should be changed to `calldata` :

- ClaimerRelayer.mintClaimerWithSig(address,uint256,uint256,address,Role,bytes).signature

Recommendation

xfu : Use `calldata` instead of `memory` for external functions where the function argument is read-only.

Client Response

Fixed

KST-18: `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	xfu

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L146
- code/Staking_TEST/contracts/KaratStakingv2.sol#L208
- code/Staking_TEST/contracts/KaratStakingv2.sol#L261

```
146:currentEpoch += 1;

208:totalKAT += amount;

261:totalKAT -= amount;
```

Description

xfu : Using the addition operator instead of plus-equals saves **113 gas**

There are `3` instances of this issue:

- should use arithmetic operator `=` replace `+=` in currentEpoch += 1
- should use arithmetic operator `=` replace `+=` in totalKAT += amount
- should use arithmetic operator `=` replace `-=` in totalKAT -= amount

Recommendation

xfu : Using arithmetic operator `=` replace assignment operator `+=` or `-=`

Client Response

Acknowledged

KST-19:Cache state variables instead of rereading

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	biakia, xfu

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L52
- code/Staking_TEST/contracts/KaratRewardv2.sol#L54
- code/Staking_TEST/contracts/KaratRewardv2.sol#L55
- code/Staking_TEST/contracts/KaratStakingv2.sol#L57
- code/Staking_TEST/contracts/KaratRewardv2.sol#L58
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L65
- code/Staking_TEST/contracts/KaratStakingv2.sol#L67
- code/Staking_TEST/contracts/KaratStakingv2.sol#L102
- code/Staking_TEST/contracts/KaratStakingv2.sol#L103
- code/Staking_TEST/contracts/KaratRewardv2.sol#L149
- code/Staking_TEST/contracts/KaratStakingv2.sol#L282-L310

```
52:uint256 public currentEpoch;

54:IERC20 private _asset;

55:IStakingContract private _staking;

57:uint256 public totalKAT;

58:mapping(address => uint256) claimerClaimedAmount;

65:IStakingContract private _staking;

67:mapping(uint256 => uint256) public totalKaratReward;

102:IERC20 public _asset;

103:CountersUpgradeable.Counter public tokenIdCounter;

149:for (uint i = 0; i < epochs.length; i++) {

282:function updatePool(
283:        uint256 validatorId,
284:        uint256 epoch
285:    ) public onlyRole(AUTHORIZED_CALLER) {
286:        require(epoch < currentEpoch, "This Epoch is not Valid");
287:        require(currentEpoch >= 1, "Must Calculate After first Day");
288:        require(!isValidatorClaimed[validatorId][epoch], "Rewards Calculated");
289:
290:        if (validatorReward[validatorId][epoch] != 0) {
291:            isValidatorClaimed[validatorId][epoch] = true;
292:            validatorEarns[validatorId] +=
293:                (validatorReward[validatorId][epoch] *
294:                    getCurrentKATReward(epoch)) /
295:                totalKaratReward[epoch];
296:
297:            //Update Pool Reward For Stakers
298:            _updateRewardIndex(
299:                validatorId,
300:                ((stakerReward[validatorId][epoch] *
301:                    getCurrentKATReward(epoch)) / totalKaratReward[epoch])
302:            );
303:        }
```

```

        require(
            epoch >= lastPoolUpdated[validatorId],
            "Cannot Update Before Days"
        );
        lastPoolUpdated[validatorId] = epoch + 1;
    }
```

Description

biakia : In function `updatePool`, the function `getCurrentKATReward` will be called twice:

```
if (validatorReward[validatorId][epoch] != 0) {
            isValidatorClaimed[validatorId][epoch] = true;
            validatorEarns[validatorId] +=
                (validatorReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) /
                totalKaratReward[epoch];

            //Update Pool Reward For Stakers
            _updateRewardIndex(
                validatorId,
                ((stakerReward[validatorId][epoch] *
                    getCurrentKATReward(epoch)) / totalKaratReward[epoch])
            );
        }
```

It is better to cache the result of the function `getCurrentKATReward` to save gas.

xfu : The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable code/Staking_TEST/contracts/addresses.

More detail see this.

There are `15` instances of this issue:

- StakedKaratPoolToken.currentEpoch should be cached with local memory-based variable in StakedKaratPoolToken.updatePool(uint256,uint256), It is called more than once:
  - require(bool,string)(epoch < currentEpoch,This Epoch is not Valid)
  - require(bool,string)(currentEpoch >= 1,Must Calculate After first Day)
- StakedKaratPoolToken.currentEpoch should be cached with local memory-based variable in StakedKaratPoolToken.tryUpdateSnapshot(address), It is called more than once:
  - startEpochTime = (currentEpoch + 1) * 86400 + firstDAYUnix
  - block.timestamp >= startEpochTime && ifEverydayRewardClaimed[currentEpoch + 1] == address(0)
- StakedKaratPoolToken.currentEpoch should be cached with local memory-based variable in StakedKaratPoolToken._snapshot(), It is called more than once:
  - currentEpoch

  - SnapShotTaken(currentEpoch)
- StakedKaratPoolToken.currentEpoch should be cached with local memory-based variable in StakedKaratPoolToken.tryUpdatePool(uint256), It is called more than once:
  - currentEpoch > 0 && lastUpdate <= currentEpoch - 1
  - i < currentEpoch
- RewardDistributor._asset should be cached with local memory-based variable in RewardDistributor._claimAndBurn(address,uint256), It is called more than once:
  - SafeERC20.safeTransfer(_asset,to,realAmount)
  - ERC20Burnable(address(_asset)).burn(burnAmount)
- RewardDistributor._staking should be cached with local memory-based variable in RewardDistributor.claimLieutenant(uint256[],uint256,uint256), It is called more than once:
  - totalRewards += _staking.getLieutenantRewardbyEpoch(msg.sender,epochs[i])
  - totalRewards += _staking.getLieutenantRewardbyEpoch(msg.sender,i_scope_0)
- RewardDistributor._staking should be cached with local memory-based variable in RewardDistributor.claimStaker(uint256,uint256), It is called more than once:
  - reward = _staking.calStaker(stakerTokenId,validatorId)
  - require(bool,string)(_staking.ownerOf(stakerTokenId) == msg.sender,Not Owner of this Token)
- StakedKaratPoolToken.totalKAT should be cached with local memory-based variable in StakedKaratPoolToken.updateClaimerReward(uint256,address,address,uint256), It is called more than once:
  - totalKAT != 0
  - poolWeightReward = (karatScore * poolBalance[validatorId]) / totalKAT / 3
- RewardDistributor.claimerClaimedAmount should be cached with local memory-based variable in RewardDistributor.claimClaimer(uint256), It is called more than once:
  - reward = totalClaimableAmount - claimerClaimedAmount[msg.sender]
  - require(bool,string)(claimerClaimedAmount[msg.sender] < totalClaimableAmount,Already Claimed)
- ClaimerRelayer._staking should be cached with local memory-based variable in ClaimerRelayer.mintClaimerWithSig(address,uint256,uint256,address,Role,bytes), It is called more than once:
  - result = _staking.tryUpdateSnapshot(to)
  - _staking.tryUpdatePool(validatorTokenId)
  - _staking.updateClaimerReward(validatorTokenId,to,lieutenantAddr,karatScore)
- ClaimerRelayer._staking should be cached with local memory-based variable in ClaimerRelayer.mintClaimer(address,uint256,uint256,address,Role), It is called more than once:
  - _staking.tryUpdatePool(validatorTokenId)
  - result = _staking.tryUpdateSnapshot(to)
  - _staking.updateClaimerReward(validatorTokenId,to,lieutenantAddr,karatScore)
- StakedKaratPoolToken.totalKaratReward should be cached with local memory-based variable in StakedKaratPoolToken.getLieutenantRewardbyEpoch(address,uint256), It is called more than once:
  - totalKaratReward[epoch] != 0
  - (lieutenantReward[lieutenantAddr][epoch] * getCurrentKATReward(epoch)) / totalKaratReward[epoch]
- StakedKaratPoolToken.totalKaratReward should be cached with local memory-based variable in StakedKaratPoolToken.getClaimerRewardbyEpoch(address,uint256), It is called more than once:

  - (claimerReward[claimer][epoch] * getCurrentKATReward(epoch)) / totalKaratReward[epoch]
- StakedKaratPoolToken._asset should be cached with local memory-based variable in StakedKaratPoolToken.unstake(uint256,uint256), It is called more than once:
  - SafeERC20.safeTransfer(_asset,msg.sender,amount - burnedAmount)
  - ERC20Burnable(address(_asset)).burn(burnedAmount)
- StakedKaratPoolToken.tokenIdCounter should be cached with local memory-based variable in StakedKaratPoolToken.safeMint(address,uint256,uint256), It is called more than once:
  - tokenId = tokenIdCounter.current()
  - tokenIdCounter.increment()

  xfu : The overheads outlined below are *PER LOOP*, excluding the first loop
- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset. More detail optimization see this

There is `1` instance of this issue:

- i < epochs.length `<array>.length` should be cached.

Recommendation

biakia : Consider caching the result of the function `getCurrentKATReward` to save gas:

```
if (validatorReward[validatorId][epoch] != 0) {
            uint256 katRewards = getCurrentKATReward(epoch);
            isValidatorClaimed[validatorId][epoch] = true;
            validatorEarns[validatorId] +=
                (validatorReward[validatorId][epoch] * katRewards) /
                totalKaratReward[epoch];

            //Update Pool Reward For Stakers
            _updateRewardIndex(
                validatorId,
                ((stakerReward[validatorId][epoch] * katRewards) / totalKaratReward[epoch])
            );
        }
```

xfu : Cache storage-based state variables in local memory-based variables appropriately to convert SLOADs to MLOADs and reduce gas consumption from 100 units to 3 units. than once for a function
xfu : Caching the `<array>.length` for the loop condition, for example:

```
// gas save (-230)
function loopArray_cached(uint256[] calldata ns) public returns (uint256 sum) {
    uint256 length = ns.length;
    for(uint256 i = 0; i < length;) {
        sum += ns[i];
        unchecked {
            i++;
        }
    }
}
```

Client Response

Acknowledged

KST-20:Use indexed events for value types as they are less costly compared to non-indexed ones

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	xfu

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L38-L43
- code/Staking_TEST/contracts/KaratStakingv2.sol#L44-L49
- code/Staking_TEST/contracts/KaratRewardv2.sol#L49
- code/Staking_TEST/contracts/KaratRewardv2.sol#L50
- code/Staking_TEST/contracts/KaratStakingv2.sol#L50
- code/Staking_TEST/contracts/KaratRewardv2.sol#L51
- code/Staking_TEST/contracts/KaratRewardv2.sol#L52

```
38:event TokenStaked(
39:        address staker,
40:        uint256 valiadtorId,
41:        uint256 stakerTokenId,
42:        uint256 amount
43:    );

44:event TokenUnstaked(
45:        address staker,
46:        uint256 valiadtorId,
47:        uint256 stakerTokenId,
48:        uint256 amount
49:    );

49:event RewardClaimerClaimed(address claimer, uint256 amount, uint256 epoch);

50:event RewardValidatorClaimed(uint256 validatorId, uint256 amount);

50:event SnapShotTaken(uint256 epoch);

51:event RewardLieutenantClaimed(address lieutenantAddr, uint256 amount);

52:event RewardStakerClaimed(address staker, uint256 amount);
```

Description

xfu : Using the `indexed` keyword for value types (`bool/int/address/string/bytes`) saves gas costs, as seen in this example.

However, this is only the case for value types, whereas indexing reference types (`array/struct`) are more expensive than their unindexed version.

Recommendation

xfu : Using the `indexed` keyword for values types `bool/int/address/string/bytes` in event

Client Response

Fixed

KST-21:Conformance to Solidity naming conventions

Category	Severity	Client Response	Contributor
Language Specific	Informational	Acknowledged	xfu, yekong

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L100
- code/Staking_TEST/contracts/KaratRewardv2.sol#L64
- code/Staking_TEST/contracts/KaratStakingv2.sol#L102

```
64:uint256 public constant startingReward = 5000000 * MULTIPLIER;

100:uint256 public constant startingReward = 5000000 * MULTIPLIER;

102:IERC20 public _asset;
```

Description

xfu : Solidity defines a naming convention that should be followed.

Rule exceptions

- Allow constant variable name/symbol/decimals to be lowercase (`ERC20`).
- Allow `_` at the beginning of the `mixed_case` match for private variables and unused parameters.

There is `1` instance of this issue:

- Variable StakedKaratPoolToken._asset is not in mixedCase
  yekong : In Solidity, it is a common best practice to name constant variables using the UPPER_SNAKE_CASE convention. This convention makes the code more readable and immediately recognizable as a constant.

Recommendation

xfu : Follow the Solidity naming convention.

For example:

```
    // contracts/KaratStakingv2.sol#L102
    // IERC20 public _asset;
    IERC20 public asset;
```

yekong : Consider renaming `startingReward` to `STARTING_REWARD`

Client Response

Acknowledged

KST-22:Logic Error in stake Function

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	yekong

Code Reference

- code/Staking_TEST/contracts/KaratStakingv2.sol#L219-L238

```
219:function stake(
220:        uint256 amount,
221:        uint256 validatorId
222:    ) public nonReentrant whenNotPaused {
223:        require(validatorId != 0, "Pool 0 is open for staking");
224:        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
225:        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
226:        safeMint(msg.sender, validatorId, amount);
227:    }
228:
229:    function stakeTo(
230:        address recipient,
231:        uint256 amount,
232:        uint256 validatorId
233:    ) public nonReentrant whenNotPaused {
234:        require(validatorId != 0, "Pool 0 is open for staking");
235:        require(amount >= minimumToStake, "Stake Amount Must Exceed Minimum");
236:        SafeERC20.safeTransferFrom(_asset, msg.sender, address(this), amount);
237:        safeMint(recipient, validatorId, amount);
238:    }
```

Description

yekong : The stake function contains a logic error in the check for validatorId. The current check implies that pool 0 is open for staking, which may not be the intended behavior.

Recommendation

yekong : Update the require statement to reflect the correct logic. If you intend to disallow staking in pool 0, you can use the following code snippet for reference:

```
require(validatorId != 0, "Pool 0 is not open for staking");
```

Client Response

Acknowledged

KST-23:Public function that could be declared external

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Declined	biakia, xfu

Code Reference

- code/Staking_TEST/contracts/ClaimerRelayer.sol#L130-L158
- code/Staking_TEST/contracts/KaratRewardv2.sol#L142-L179

```
function mintClaimerWithSig(
        address to,
        uint256 validatorTokenId,
        uint256 karatScore,
        address lieutenantAddr,
        Role role,
        bytes memory signature
    ) public {
        _claimer.mintClaimerwithSig(
            to,
            validatorTokenId,
            karatScore,
            lieutenantAddr,
            role,
            signature
        );

        bool result = _staking.tryUpdateSnapshot(to);
        if (result) {
            SafeERC20.safeTransfer(_katToken, to, rewardAmount);
        }
        _staking.tryUpdatePool(validatorTokenId);
        _staking.updateClaimerReward(
            validatorTokenId,
            to,
            lieutenantAddr,
            karatScore
        );
    }

function claimLieutenant(
        uint256[] calldata epochs,
        uint256 from,
        uint256 to
    ) public whenNotPaused {
        uint256 totalRewards;
        if (from == to && from == 0) {
            for (uint i = 0; i < epochs.length; i++) {
                require(
                    !isLieutenantClaimed[msg.sender][epochs[i]],
                    "Rewards Calculated"
                );

                isLieutenantClaimed[msg.sender][epochs[i]] = true;
                totalRewards += _staking.getLieutenantRewardbyEpoch(
                    msg.sender,
                    epochs[i]
                );
            }
        } else {
            require(from < to, "NOT Valid");
            for (uint i = from; i < to; i++) {
                require(
                    !isLieutenantClaimed[msg.sender][i],
                    "Rewards Calculated"
                );

                isLieutenantClaimed[msg.sender][i] = true;
                totalRewards += _staking.getLieutenantRewardbyEpoch(
                    msg.sender,
                    i
                );
            }
        }
        _claimAndBurn(msg.sender, totalRewards);

        emit RewardLieutenantClaimed(msg.sender, totalRewards);
    }
```

Description

biakia : `external` functions are sometimes more efficient when they receive large arrays of data. In `public` functions, solidity immediately copies array arguments to memory, while `external` functions can read directly from calldata. Memory allocation is expensive, whereas reading from calldata is cheap.
xfu : `public` functions that are never called by the contract should be declared `external`, and its immutable parameters should be located in `calldata` to save gas.

There is `1` instance of this issue:

- mintClaimerWithSig(address,uint256,uint256,address,Role,bytes) should be declared external: - ClaimerRelayer.mintClaimerWithSig(address,uint256,uint256,address,Role,bytes) Moreover, the following function parameters should change its data location: signature location should be calldata

Recommendation

biakia : Consider using `external`:

```
function claimLieutenant(
        uint256[] calldata epochs,
        uint256 from,
        uint256 to
    ) external whenNotPaused {
    ...
```

xfu : Use the `external` attribute for functions never called from the contract, and change the location of immutable parameters to `calldata` to save gas.

Client Response

Declined

KST-24:Variables that could be declared as immutable

Category	Severity	Client Response	Contributor
Gas Optimization	**Informational**	**Declined**	**xfu, biakia**

Code Reference

- code/Staking_TEST/contracts/ClaimerRelayer.sol#L64
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L65
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L66
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L67
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L69
- code/Staking_TEST/contracts/ClaimerRelayer.sol#L64-L67
- code/Staking_TEST/contracts/KaratRewardv2.sol#L54-L56
- code/Staking_TEST/contracts/KaratStakingv2.sol#L102

```
54:IERC20 private _asset;
55:    IStakingContract private _staking;
56:    IValidatorContract private _validator;

64:IERC20 private _katToken;

64:IERC20 private _katToken;
65:    IStakingContract private _staking;
66:    IClaimerContract private _claimer;
67:    IReward private _reward;

65:IStakingContract private _staking;

66:IClaimerContract private _claimer;

67:IReward private _reward;

69:uint256 public firstDAY;

102:IERC20 public _asset;
```

Description

xfu : Avoids a Gsset (20000 gas) in the constructor, and replaces the first access in each transaction (Gcoldsload - 2100 gas) and each access thereafter (Gwarmacces - 100 gas) with a PUSH32 (3 gas).

While strings are not value types, and therefore cannot be immutable/constant if not hard-coded outside of the constructor, the same behavior can be achieved by making the current contract abstract with virtual functions for the string accessors, and having a child contract override the functions with the hard-coded implementation-specific values.

There are `5` instances of this issue:

- ClaimerRelayer._katToken should be immutable
- ClaimerRelayer._staking should be immutable
- ClaimerRelayer._claimer should be immutable
- ClaimerRelayer._reward should be immutable
- ClaimerRelayer.firstDAY should be immutable

  biakia :

```
      IERC20 private _katToken;
      IStakingContract private _staking;
      IClaimerContract private _claimer;
      IReward private _reward;
```

In contract `ClaimerRelayer`, the linked variables assigned in the constructor can be declared as immutable.

```
      IERC20 private _asset;
      IStakingContract private _staking;
      IValidatorContract private _validator;
```

In contract `RewardDistributor`, the linked variables assigned in the constructor can be declared as immutable.

```
  IERC20 public _asset;
```

In contract `StakedKaratPoolToken`, the linked variables assigned in the constructor can be declared as immutable.

Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

xfu : Add the `immutable` attribute to state variables that never change or are set only in the constructor.
biakia : We recommend declaring these variables as immutable. Please note that the immutable keyword only works in Solidity version v0.6.5 and up.

Client Response

Declined

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.