# AI Arena QA Audit Report

Lead Auditors:

- Benas Volkovas

## Table of Contents

## Disclaimer

I make 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 audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Issues found

| Severity | Number of issues found |
|---|---|
| Low | 3 |
| Info | 1 |
| Total | 4 |

## Findings

### Low severity

#### [L-01] `MergingPool::getFighterPoints` can only return points for token id 0, which makes the function redundant and misleading

**Description:** The `MergingPool::getFighterPoints` function is expected to return an array of points corresponding to the fighters' token IDs up to the specified maximum token ID. However, the return array `points` is only initialized with length of 1. If the `maxId` is greater than 1, the function will revert making it impossible to retrieve points for multiple fighters.

Also, the natspec documentation for the function is misleading as it states that `maxId` is the maximum token ID up to which the points will be retrieved. However, the loop inside only iterates up to `maxId - 1` and not including `maxId` .

```
    function getFighterPoints(uint256 maxId) public view returns(uint256[] memory) {
@>      uint256[] memory points = new uint256[](1);
@>      for (uint256 i = 0; i < maxId; i++) {
            points[i] = fighterPoints[i];
        }
        return points;
    }
```

Reference: MergingPool.sol#L206-207

**Impact:** Unusable function and misleading documentation.

**Proof of Concept:** Paste this test in `MergingPool.t.sol` :

```
function test_audit_RevertsIfTryingToGet2FighterPoints() public {
    // owner mints a fighter by claiming
    _mintFromMergingPool(_ownerAddress);
    _mintFromMergingPool(_ownerAddress);
    assertEq(_fighterFarmContract.ownerOf(0), _ownerAddress);
    assertEq(_fighterFarmContract.ownerOf(1), _ownerAddress);

    // rankedeBattle contract adds points
    vm.startPrank(address(_rankedBattleContract));
    _mergingPoolContract.addPoints(0, 100);
    _mergingPoolContract.addPoints(1, 200);
    vm.stopPrank();

    assertEq(_mergingPoolContract.totalPoints(), 300);

    // getFighterPoints for owners fighter
    uint256 maxId = 2;
    vm.expectRevert(stdError.indexOOBError);
    _mergingPoolContract.getFighterPoints(maxId);
}
```

**Recommended Mitigation:** Update the `getFighterPoints` function to return an array of points corresponding to the fighters' token IDs up to the specified maximum token ID. Use `maxId` to initialize the `points` array length. Also, update loop to iterate up to `maxId` and not `maxId - 1` .

```
    function getFighterPoints(
        uint256 maxId
    ) public view returns (uint256[] memory) {
-       uint256[] memory points = new uint256[](1);
+       uint256[] memory points = new uint256[](maxId + 1);
-       for (uint256 i = 0; i < maxId; i++) {
+       for (uint256 i = 0; i <= maxId; i++) {
            points[i] = fighterPoints[i];
        }
        return points;
    }
```

## [L-02] `Neuron::constructor` doesn't setup roles for all actors in the system, which can lead to unauthorized access for some functions

**Description:** The `constructor` function doesn't call the `setupRoles` function to setup roles for all actors in the system. This can lead to unauthorized access for some functions. Also, `_setupRole` function is deprecated in favor of `_grantRole` , which should be called in the constructor. Additionally, noone is granted the `DEFAULT_ADMIN_ROLE` role, which is required to grant roles to other actors and to perform administrative tasks. In `constructor` function `MINTER_ROLE` , `SPENDER_ROLE` , `STAKER_ROLE` , `DEFAULT_ADMIN_ROLE` roles are not setup.

```
    constructor(
        address ownerAddress,
        address treasuryAddress_,
        address contributorAddress
    ) ERC20("Neuron", "NRN") {
        _ownerAddress = ownerAddress;
        treasuryAddress = treasuryAddress_;
        isAdmin[_ownerAddress] = true;
        _mint(treasuryAddress, INITIAL_TREASURY_MINT);
        _mint(contributorAddress, INITIAL_CONTRIBUTOR_MINT);
    }
```

Reference: Neuron.sol#L68

**Impact:** Some actors might be blocked from accessing some functions, and the contract might not be able to perform administrative tasks for access control.

**Proof of Concept:**

1. Admin deploys the contract
2. Admin forgets to call `addMinter` , `addSpender` and `addStaker` functions to setup roles for all actors in the system.
3. RankedBattle, GameItems, FighterFarm contracts are deployed and they are not able to call functions that require `MINTER_ROLE` , `SPENDER_ROLE` and `STAKER_ROLE` roles.

**Recommended Mitigation:** Make `addMinter` , `addSpender` and `addStaker` functions public and call them in the constructor to setup roles for all actors in the system. Also, don't forget to grant `DEFAULT_ADMIN_ROLE` role to the contract deployer.

```
    constructor(
        address ownerAddress,
        address treasuryAddress_,
        address contributorAddress,
+       address rankedBattleAddress,
+       address gameItemsAddress,
+       address fighterFarmAddress
    ) ERC20("Neuron", "NRN") {
        _ownerAddress = ownerAddress;
        treasuryAddress = treasuryAddress_;
        isAdmin[_ownerAddress] = true;

+       _grantRole(DEFAULT_ADMIN_ROLE, _ownerAddress);
+       neuron.addMinter(rankedBattleAddress);
+       neuron.addStaker(rankedBattleAddress);
+       neuron.addSpender(gameItemsAddress);
+       neuron.addSpender(fighterFarmAddress);

        _mint(treasuryAddress, INITIAL_TREASURY_MINT);
        _mint(contributorAddress, INITIAL_CONTRIBUTOR_MINT);
    }
```

## [L-03] `Neuron::mint` function doesn't allow to mint all `MAX_SUPPLY` quantity of `NRN` tokens, which makes the protocol to not work as expected

**Description:** In `mint` function there is a `require` statement that checks if the `totalSupply` with `amount` that caller wants to mint is less than `MAX_SUPPLY`. This ensures that only `MAX_SUPPLY - 1` tokens can ever be minted. This is not expected behavior and the `mint` function should allow to mint all `MAX_SUPPLY` quantity of `NRN` tokens.

```
    require(
@>      totalSupply() + amount < MAX_SUPPLY,
        "Trying to mint more than the max supply"
    );
```

Reference: Neuron.sol#L156

**Impact:** The protocol will not work as expected and the `mint` function will not allow to mint all `MAX_SUPPLY` quantity of `NRN` tokens.

**Proof of Concept:** Paste this test in `Neuron.t.sol`:

```
function test_audit_CannotMintMaxSupplyTokens() public {
    address minter = makeAddr("minter");
    uint256 maxSupply = _neuronContract.MAX_SUPPLY() -
        _neuronContract.INITIAL_TREASURY_MINT() -
        _neuronContract.INITIAL_CONTRIBUTOR_MINT();
    _neuronContract.addMinter(minter);

    vm.expectRevert();
    vm.prank(minter);
    _neuronContract.mint(minter, maxSupply);
}
```

**Recommended Mitigation:** Update the require statement to use `<=` instead of `<` to allow to mint all `MAX_SUPPLY` quantity of `NRN` tokens.

```
    require(
-        totalSupply() + amount < MAX_SUPPLY,
+        totalSupply() + amount <= MAX_SUPPLY,
        "Trying to mint more than the max supply"
    );
```

## Informational

### [I-01] No checks for `iconsTypes` array length, which can lead to out of bounds access

**Description:** In `FighterFarm::redeemMintPass` function, the `iconsTypes` array is accessed without any checks for its length. However, the `iconsTypes` array is expected to be of length of the other parameters passed to the function. This can lead to out of bounds access and can cause the contract to revert.

```
    function redeemMintPass(
        uint256[] calldata mintpassIdsToBurn,
        uint8[] calldata fighterTypes,
        uint8[] calldata iconsTypes,
        string[] calldata mintPassDnas,
        string[] calldata modelHashes,
        string[] calldata modelTypes
    ) external {
        require(
            mintpassIdsToBurn.length == mintPassDnas.length &&
                mintPassDnas.length == fighterTypes.length &&
                fighterTypes.length == modelHashes.length &&
                modelHashes.length == modelTypes.length
        );
```

Reference: [FighterFarm.sol#L243-L248]()

**Impact:** This can lead to out of bounds access and can cause the contract to revert.

**Proof of Concept:**

Paste this test in `FighterFarm.t.sol` :

```
function test_audit_RevertsIfIconsTypesArrayLengthTooShor() public {
    uint8[2] memory numToMint = [1, 0];
    bytes memory signature = abi.encodePacked(
        hex"20d5c3e5c6b1457ee95bb5ba0cbf35d70789bad27d94902c67ec738d18f665d84e316edf9b23c154054c7
    );
    string[] memory _tokenURIs = new string[](1);
    _tokenURIs[
        0
    ] = "ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";

    _mintPassContract.claimMintPass(numToMint, signature, _tokenURIs);

    // once owning one i can then redeem it for a fighter
    uint256[] memory _mintpassIdsToBurn = new uint256[](1);
    string[] memory _mintPassDNAs = new string[](1);
    uint8[] memory _fighterTypes = new uint8[](1);
    uint8[] memory _iconsTypes = new uint8[](0);
    string[] memory _neuralNetHashes = new string[](1);
    string[] memory _modelTypes = new string[](1);

    _mintpassIdsToBurn[0] = 1;
    _mintPassDNAs[0] = "dna";
    _fighterTypes[0] = 0;
    _neuralNetHashes[0] = "neuralnethash";
    _modelTypes[0] = "original";

    // approve the fighterfarm contract to burn the mintpass
    _mintPassContract.approve(address(_fighterFarmContract), 1);

    vm.expectRevert();
    _fighterFarmContract.redeemMintPass(
        _mintpassIdsToBurn,
        _fighterTypes,
        _iconsTypes,
        _mintPassDNAs,
        _neuralNetHashes,
        _modelTypes
    );
}
```

**Recommended Mitigation:** Add a require statement to check the length of `iconsTypes` array.

```
    function redeemMintPass(
        uint256[] calldata mintpassIdsToBurn,
        uint8[] calldata fighterTypes,
        uint8[] calldata iconsTypes,
        string[] calldata mintPassDnas,
        string[] calldata modelHashes,
        string[] calldata modelTypes
    ) external {
        require(
            mintpassIdsToBurn.length == mintPassDnas.length &&
                mintPassDnas.length == fighterTypes.length &&
                fighterTypes.length == modelHashes.length &&
+                   modelHashes.length == modelTypes.length &&
+                   modelTypes.length == iconsTypes.length
        );
    }
```