Protocol Audit Report

Version 1.0

Quaternion

August 7, 2024

Protocol Audit Report

Quaternion

August 7, 2024

Prepared by: Quaternion Lead Auditors: - Quaternion

Table of Contents

Protocol Summary

TraitForge is a competitive and collaborative breeding game with a nuke function. Each NFT (Entity), has set Traits, created and calculated before mint, by indexing the entropy slots. Each digit or total entropy, gives the NFT its unqique traits, that are determined but completely dispursed/random. Players can forge (breed), by either listing a forger, or paying a forger and receiving the new NFT. Nuking is a function where players burn their NFT, in return claim a portion of ETH from the nuke-fund, in proportion to the 'nuke factor' of the entity, which increases overtime.

Players compete in every aspect of the game, whether its minting the best NFT's available, trading or listing for forging in a competitive market or nuking when the time is right.

Disclaimer

The Quaternion team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

Audit Details

Scope

File	Logic Contracts	Interfaces	nSLOC	Purpose	Libraries used
/contracts/DevFund/DevFund.sol	1	****	77		@openzeppelin/contracts/access/O
/contracts/EntityForging/EntityForging.sol	1	****	152		@openzeppelin/contracts/security/
/contracts/EntityTrading/EntityTrading.sol	1	****	82		@openzeppelin/contracts/token/ER
/contracts/EntropyGenerator/EntropyGenerator.sol	1	****	147		@openzeppelin/contracts/access/O
/contracts/NukeFund/NukeFund.sol	1	****	150		@openzeppelin/contracts/access/O
/contracts/TraitForgeNft/TraitForgeNft.sol	1	****	272		@openzeppelin/contracts/token/ER
Totals	**6**	****	**880**		

Roles

Role	Description
Contract Owner	

Issues found

Medium

[M-1] Checking for occurence of "Golden god" entropy in `EntropyGenerator::writeEntropyBatch1` and `EntropyGenerator::writeEntropyBatch2` is incorrect and not present in `EntropyGenerator::writeEntropyBatch3`

Description The `EntropyGenerator::writeEntropyBatch2` and `EntropyGenerator::writeEntropyBatch1` fill the entropy slots with pseudorandom values. While doing so, they only check that the pseudorandom value filled in the entropy_slot is not equal to `999_999` which is wrong since this allow values which would give multiple token holders in each generation golden entropy and access to 50% of the nukeFund.

Impact The function `EntropyGenerator::writeEntropyBatch3` allows for a chance where frontrunning can cause block.number for execution to be such that the value in the slots can contain multiple 999,999 which is the golden gold allowing 50% nukefund retrieval to multiple users draining funds from the NukeFund.

The function `EntropyGenerator::writeEntropyBatch2` although checks that the pseudorandom value is not 999_999 in any slot. But if the pseudorandom value is like 999_999_000_000, 999_999_999_999, then the value would be inserted in the slot as it not equal to 999_999. This would result in multiple entries getting "Golden god" in the same generation

Proof of Concept 1. The occurrence of 999,999 in a single slot is unlikely but an ideal block number can allow for the occurrence of "the golden gold" in multiple slots.

Tools Used Manual Review

Recommended Mitigation 1. Check for the values that come into the slots, such that no slot can contain 999,999 or any such integer that contains 999,999 in a subslot. Following function (checkNoGoldenGoldInAnySlot) can be used inside EntropyGenerator contract

```
  function writeEntropyBatch2() public {
    require(
      lastInitializedIndex >= batchSize1 && lastInitializedIndex <
          batchSize2,
      'Batch 2 not ready or already initialized.'
    );

    uint256 endIndex = lastInitializedIndex + batchSize1;
    unchecked {
      for (uint256 i = lastInitializedIndex; i < endIndex; i++) {
        uint256 pseudoRandomValue = uint256(
          keccak256(abi.encodePacked(block.number, i))
        ) % uint256(10) ** 78;
-       require(pseudoRandomValue != 999_999, 'Invalid value, retry.');
+       require(checkNoGoldenGoldInAnySlot(pseudoRandomValue), 'Invalid
    value, retry');
        entropySlots[i] = pseudoRandomValue;
      }
    }
    lastInitializedIndex = endIndex;
  }


  function writeEntropyBatch3() public {
    require(
      lastInitializedIndex >= batchSize2 && lastInitializedIndex <
          maxSlotIndex,
      'Batch 3 not ready or already completed.'
    );
    unchecked {
      for (uint256 i = lastInitializedIndex; i < maxSlotIndex; i++) {
        uint256 pseudoRandomValue = uint256(
          keccak256(abi.encodePacked(block.number, i))
        ) % uint256(10) ** 78;
+       require(checkNoGoldenGoldInAnySlot(pseudoRandomValue), 'Invalid
    value, retry');
```

```
          entropySlots[i] = pseudoRandomValue;
        }
      }
      lastInitializedIndex = maxSlotIndex;
    }

+   function checkNoGoldenGoldInAnySlot(uint256 value) public returns(
      bool) {
+     for(uint i = 0 ; i<13 ; i++){
+       uint256 slotValue = value % (1_000_000);
+       if(slotValue == 999_999) return false;
+       value /= 1_000_000;
+     }
+     return true;
    }
```

Low

[L-1] Pushing funds directly to forger accounts may waste gas if user contract reverts on receive

Description: Whenever a merger token owner calls `EntityForging::forgeWithListed` the forgerShare is directly sent to the forger. This could revert the call as user may use contract and has receive function revert with error. Link

Impact: The impact of this is low, since user will be wasting small amount of gas since reverts are not for free.

Recommended Mitigation: 1. Allow push over pull mechanism for allowing forgers to withdraw funds from smart contract.

```
contract EntityForging is IEntityForging, ReentrancyGuard, Ownable,
    Pausable {
.
.
.
+    mapping(address forgerAddress => uint256 forgerFeesCollected)
    private s_forger_to_fees_accrued;
.
.
.
     function forgeWithListed( uint256 forgerTokenId, uint256
         mergerTokenId){
.
.
.
-         (bool success_forge, ) = forgerOwner.call{ value: forgerShare
    }('');
-        require(success_forge, 'Failed to send to Forge Owner');
+        s_forger_to_fees_accrued[forgeOwner] += forgerShare;
     }

+    function withdrawFees() public payable {
+        uint256 amount = s_forger_to_fees_accrued[msg.sender];
+        s_forger_to_fees_accrued[forgeOwner] = 0;
+        (bool success, ) = payable(msg.sender).call{value: amount}("");
+        require(success_forge, 'Eth withdrawal failed');
     }
.
.
.
}
```

[L-2] `EntropyGenerator::deriveTokenParameters` outputs wrong nukeFactor value

Lines of code

https://github.com/code-423n4/2024-07-traitforge/blob/279b2887e3d38bc219a05d332cbcb0655b2dc644/contracts/En

Description The `EntropyGenerator::deriveTokenParameters` wrongly informs the users of the nuking potential of Nfts by always giving nuking factor as 0.

Impact This would incorrectly inform the users of their initialNukeFactor, and eventually the potential amount of funds they can accrue in the nuking of NFT.

Proof of Concept paste the following code in describe test of `EntropyGenerator.test.ts`

```
  it("should check that nuke factor is always zero", async function(){
    let cnt = 0 ;
    for(let slotIndex = 0 ; slotIndex<700 ; slotIndex++){
      for(let numberIndex = 0; numberIndex<12 ; numberIndex++ ){
        const [nukeFactor, x, y, z] = await entropyGenerator.
            deriveTokenParameters(slotIndex, numberIndex);
        if(nukeFactor == 0n) cnt++;
      }
    }
    console.log(cnt);
  })
```

The count here will be 8400

Tools Used Manual Review, Hardhat Test

Recommended Mitigation Steps 1. correct the number of trailing zeroes while calculating the nukeFactor.

```
    function deriveTokenParameters(
    uint256 slotIndex,
    uint256 numberIndex
  )
    public
    view
    returns (
      uint256 nukeFactor,
      uint256 forgePotential,
      uint256 performanceFactor,
      bool isForger
    )
  {
    uint256 entropy = getEntropy(slotIndex, numberIndex);

    // example calcualtions using entropyto derive game-related
       parameters
-    nukeFactor = entropy / 4000000;
+    nukeFactor = entropy / 40;
.
.
.
    return (nukeFactor, forgePotential, performanceFactor, isForger);
  }
```

Assessed type

Math