One-Shot Audit Report

Version 1.0

Owen Lee

March 8, 2024

One-Shot Audit Report

Owen Lee

March 8, 2024

Prepared by: Owen Lee. Lead Security Researcher: - Owen Lee.

Table of Contents

Protocol Summary

The protocol consists of a unique ecosystem called “One Shot” designed for minting, staking, and battling rapper NFTs.

Disclaimer

I, Owen Lee 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

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
Commit hash: None.
```

Scope

```
./src/
#-- CredToken.sol
#-- OneShot.sol
```

```
4    #-- RapBattle.sol
5    #-- Streets.sol
```

Roles

User - Should be able to mint a rapper, stake and unstake their rapper and go on stage/battle.

Executive Summary

The audit of the One Shot protocol revealed critical issues with randomness, reentrancy vulnerabilities, and lack of efficient validation checks, potentially leading to undesirable outcomes for users. Urgent fixes and enhancements are required to address these vulnerabilities before deployment. The team is advised to prioritize security measures and thorough testing to ensure a safe and fair protocol.

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	1
Info	0
Total	3

Findings

High

[H-1] Weak randomness in `RapBattle::_battle` allows users to influence or predict the winning contester.

Description: Hashing `msg.sender`, `block.timeStamp` and `bloc.prevrandao` together creates a predictable final number. A predicatable number is not a good random number. Malicious users

can manipulate these values or know them ahead of time to choose the winner of the Rap battle themselves.

Impact: Any user can influence the winner of the rap battle thus winning both the defender and challenger's bets. This would make the entire protocol worthless if it becomes a gas war as to who wins the rap battle.

Proof of Concept:

1. Validators can know ahead of time the block.timestamp and block.difficulty and use that to predict when/how to participate. See the solidity blog on prevrandao.
2. User can mine/manipulate their `msg.sender` value to result in their address being used to generate the winner!
3. Users can revert their `_battle` transaction if they dont like the winner.

Using on-chain values as a randomness seed is a well-documented attack vector in the blockchain space.

Recommended Mitigation: Consider using a cryptographically provable random number generator such as Chainlink VRF.

Medium

[M-1] Lack of handling for equal skill in the `RapBattle::_battle` function thus leading to an undesired outcome for contesters.

Description: The `RapBattle::_battle` function does not include logic to handle the scenario where the randomly generated `random` value is equal to `defenderRapperSkill`. This means that when `random` equals `defenderRapperSkill`, indicating a tie or draw in the battle, the contract's behavior is undefined.

```
1
2  function _battle(uint256 _tokenId, uint256 _credBet) internal {
3          address _defender = defender;
4          require(defenderBet == _credBet, "RapBattle: Bet amounts do not
               match");
5          uint256 defenderRapperSkill = getRapperSkill(defenderTokenId);
6          uint256 challengerRapperSkill = getRapperSkill(_tokenId);
7          uint256 totalBattleSkill = defenderRapperSkill +
              challengerRapperSkill;
8          uint256 totalPrize = defenderBet + _credBet;
9
10         uint256 random =
11             uint256(keccak256(abi.encodePacked(block.timestamp, block.
                   prevrandao, msg.sender))) % totalBattleSkill;
```

```
12
13
14          defender = address(0);
15          emit Battle(msg.sender, _tokenId, random < defenderRapperSkill
                ? _defender : msg.sender);
16
17          // As seen there is no explicit check if the random value is
                equal to the defenderRapperSkill, instead there is a check
                for less or equal to but even if there is a draw the
                defender still gets rewarded both bets which is quite unfair
                .
18  @>          if (random <= defenderRapperSkill) {
19
20              credToken.transfer(_defender, defenderBet);
21              credToken.transferFrom(msg.sender, _defender, _credBet);
22          } else {
23
24              credToken.transfer(msg.sender, _credBet);
25          }
26          totalPrize = 0;
27
28          oneShotNft.transferFrom(address(this), _defender,
                defenderTokenId);
29      }
```

Detail: As seen in the highlighted part of the function, there is no explicit check if the random value is equal to the defenderRapperSkill, instead there is a check for less or equal to but even if there is a draw the defender still gets rewarded both bets which is quite unfair.

Impact: The lack of handling for equal skill could lead to confusion or undesired outcomes for users such as the bearing the reward of both bets even though there was a draw.

Proof of Concept:

1. When the `_battle` function is called with the `random` equal to `defenderRapperSkill`, If both let's say are equal to `50`.
2. The function's logic does not handle the scenario, leading to an unfair outcome whereby the `defender` address gets to win the total of both bets in a draw situation.

Recommended Mitigation: Introduce logic to handle the case where `random` is equal to `defenderRapperSkill`

```
1
2   function _battle(uint256 _tokenId, uint256 _credBet) internal {
3           address _defender = defender;
4           require(defenderBet == _credBet, "RapBattle: Bet amounts do not
                match");
5           uint256 defenderRapperSkill = getRapperSkill(defenderTokenId);
6           uint256 challengerRapperSkill = getRapperSkill(_tokenId);
```

```
7           uint256 totalBattleSkill = defenderRapperSkill +
                challengerRapperSkill;
8           uint256 totalPrize = defenderBet + _credBet;
9
10          uint256 random =
11              uint256(keccak256(abi.encodePacked(block.timestamp, block.
                    prevrandao, msg.sender))) % totalBattleSkill;
12
13          defender = address(0);
14          emit Battle(msg.sender, _tokenId, random < defenderRapperSkill
                ? _defender : msg.sender);
15
16
17  -       if (random <= defenderRapperSkill) {
18
19  -           credToken.transfer(_defender, defenderBet);
20  -           credToken.transferFrom(msg.sender, _defender, _credBet);
21  -       } else {
22
23  -           credToken.transfer(msg.sender, _credBet);
24  -       }
25
26          // Handle Equal Skill
27  +   if (random == defenderRapperSkill) {
28          // Refund the bets to both players
29  +       credToken.transfer(msg.sender, _credBet);
30  +       credToken.transfer(_defender, defenderBet);
31  +   } else if (random <= defenderRapperSkill) {
32          // Defender wins
33  +       credToken.transfer(_defender, defenderBet);
34  +       credToken.transferFrom(msg.sender, _defender, _credBet);
35  +   } else {
36          // Challenger wins
37  +       credToken.transfer(msg.sender, _credBet);
38  +   }
39          totalPrize = 0;
40
41          oneShotNft.transferFrom(address(this), _defender,
                defenderTokenId);
42      }
```

Explanation:

1. Added a check for `random == defenderRapperSkill` to handle the tie scenario.
2. If `random` equals `defenderRapperSkill`, both the `defender` and the `challenger` will receive a refund of their bets. This ensures that the contract handles all possible outcomes of the battle, including ties, in a fair and defined manner.

Final Note: Implementing this mitigation ensures that the contract behavior is consistent and fair, preventing unexpected outcomes and providing a clear resolution for tie scenarios in the battle.

Low

[L-1] The `OneShot::mintRapper` function does not adhere to the CEI (Checks-Effects-Interactions) pattern.

Description: The `mintRapper` function updates contract state after calling the `safeMint` function, potentially exposing the function to a reentrancy attack.

```
function mintRapper() public {
        uint256 tokenId = _nextTokenId++;
        _safeMint(msg.sender, tokenId);

@>        rapperStats[tokenId] =
            RapperStats({weakKnees: true, heavyArms: true,
                spaghettiSweater: true, calmAndReady: false, battlesWon:
                 0});
    }
```

Impact: The impact of this vulnerability is currently low due to the absence of any funds at risk in this specific function. However, it can introduce unpredictability and potential security risks.

Recommended Mitigation: Refactor the `OneShot::mintRapper` to follow the CEI pattern, placing the state change before the external call for security and predictability.

```

function mintRapper() public {
        uint256 tokenId = _nextTokenId++;
+    rapperStats[tokenId] = RapperStats({weakKnees: true, heavyArms:
    true, spaghettiSweater: true,calmAndReady: false, battlesWon: 0});

      _safeMint(msg.sender, tokenId);

-     rapperStats[tokenId] = RapperStats({weakKnees: true, heavyArms:
    true, spaghettiSweater: true, calmAndReady: false, battlesWon: 0});
    }
```