
BELLUM
GALAXY

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
  - Methodology
  - Audit Findings
    - High Severity Vulnerabilities
    - Medium Severity Vulnerabilities
    - Minor Observations
  - Conclusion
  - Appendices

Protocol Summary

Disclaimer

The Bellum Galaxy 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

- **Project Name:**

    - OneShot

- **Smart Contract Address:**
    - Not deployed.
- **Audit Date:**
    - 24/02/2024
- **Audit Tools Used:**
    - Code Review
    - Solidity Code Metrics
    - Slither
    - Aderyn
- **Auditors:**
    - Barba

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

Issues found

Severity	Number of issues found
High	8
Medium	2
Low	1
Gas	4
Info	1
Total	16

Audit Findings

High Severity Vulnerabilities

- **DoS - A malicious user can call `RapBattle.sol::goOnStageOrBattle` and lock other users out of the Stage**
  - **Description:**
    - The protocol design allows only one battle at a time. So, a user can battle himself by calling multiple times the `RapBattle::goOnStageOrBattle` function leading to a pump in his skills and locking the other users out. Because there isn't a penalty to the loser, besides the bet value.
  - **Impact:**
    - Break the Battle funcionality.
  - **Proof of Concept:**

    ▶ Add the following code to `OneShotTest.t.sol`

    ```
      function testPoCGoOnStage() public mintRapper {
          vm.startPrank(user);
          oneShot.approve(address(rapBattle), 0);
          rapBattle.goOnStageOrBattle(0, 0);
          address defender = rapBattle.defender();
          assert(defender == address(user));

          rapBattle.goOnStageOrBattle(0, 1);
      }
    ```

  - **Recommendation:**

    ▶ See the code recommendation below

    ```
      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 {
    +         if(msg.sender == defender){
    +           revert RapBattle__YouCantBattleYourself();
    +         }
              //!!! access control - Lack of tokenId validation. Anyone can
    ```

```
call using other people tokenId.
          // credToken.transferFrom(msg.sender, address(this),
_credBet);
          _battle(_tokenId, _credBet);
      }
  }
```

- **No bet value check on `RapBattle::goOnStageOrBattle`, leading to bets with a value of 0.**
  - **Description:**
    - The protocol design allows only one battle at a time. So, a user can call `RapBattle::goOnStageOrBattle` constantly betting 0 Cred and blocking other users to battle
  - **Impact:**
    - Break the Rap Battle funcionality in the `RapBattle::goOnStageOrBattle` function.
  - **Proof of Concept:**

    ▶ Add the following code to `OneShotTest.t.sol`

    ```
      function testPoCGoOnStage() public mintRapper {
          vm.startPrank(user);
          oneShot.approve(address(rapBattle), 0);
          rapBattle.goOnStageOrBattle(0, 0);
          address defender = rapBattle.defender();
          assert(defender == address(user));
      }
    ```

  - **Recommendation:**

    ▶ See the code recommendation below

    ```
      function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet)
    external {
    +     if(_credBet < 1){
    +       revert RapBattle__YouMustPutYourCredInTheLine();
    +     }
          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 {
    ```

```
            //!!! access control - Lack of tokenId validation. Anyone can
call using other people tokenId.
            // credToken.transferFrom(msg.sender, address(this),
_credBet);
            _battle(_tokenId, _credBet);
        }
    }
```

- **User can call `RapBattle::goOnStageOrBattle` with other people NFT or Non existent NFT, breaking the protocol design**
  - **Description:**
    - A user can call `RapBattle::goOnStageOrBattle` as a challenger passing a non-existent NFT ID or using other people's NFT ID to battle.
  - **Impact:**
    - The user will be able to collect Cred from the battle without having a Rapper NFT.
  - **Proof of Concept:**
    - Add the code below to `OneShotTest.t.sol`
    - Call the test by using `forge test --mt testPoCGoOnStage -vvvvv`
    - The function will fo through, however you will receive the error `FAIL. Reason: ERC721NonexistentToken(10)` from the `ownerOf` function after the execution.

      ▶ Add the code below to `OneShotTest.t.sol`

```
    function testIfCanCallWithNonExistantNFTID() public
twoSkilledRappers {
        vm.startPrank(user);
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 10);
        rapBattle.goOnStageOrBattle(0, 3);
        vm.stopPrank();

        vm.startPrank(challenger);
        cred.approve(address(rapBattle), 10);

        rapBattle.goOnStageOrBattle(100, 3);
        vm.stopPrank();

        assert(oneShot.ownerOf(0) == address(user));
  @>    address nftOwner = oneShot.ownerOf(100);
    }
```

  - **Recommendation:**

▶ See the code recommendation below

```
  function goOnStageOrBattle(uint256 _tokenId, uint256 _credBet)
external {
+     if(oneShotNft.ownerOf(_tokenId) != msg.sender){
+         revert RapBattle__YouMustBeTheNFTOwner();
+     }
      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);
      }
  }
```

- **The Challenger user can challenge the Defender without having the Cred balance, leading to risk zero and no reward to the Defender if he won.**
  - **Description:**
    - `RapBattle::goOnStageOrBattle` has no balance verification. In this scenario, the challenger doesn't need to have `CredToken` to call the function.
  - **Impact:**
    - The defender will not receive rewards if he wins the battle. If the challenger wins, the defender still loses cred.
  - **Proof of Concept:**

▶ Add the code below to `OneShotTest.t.sol`

```
    function testPoCGoOnStage() public mintRapper {
        vm.startPrank(address(streets));
        cred.mint(user, 10);
        vm.stopPrank();

        vm.startPrank(user);
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 10);
        rapBattle.goOnStageOrBattle(0, 10);
        address defender = rapBattle.defender();
        vm.stopPrank();
```

```
        assert(defender == address(user));
        uint256 slimBalance = cred.balanceOf(challenger);
        vm.startPrank(challenger);
        console.log(slimBalance);
        rapBattle.goOnStageOrBattle(0, 10);
    }
```

- **Recommendation:**

▶ See the recommendation code below

```
  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 {
+          if(credToken.balanceOf(msg.sender) < defenderBet){
+              revert RapBattle__YouDontHaveEnoughFunds();
+          }
          // credToken.transferFrom(msg.sender, address(this),
_credBet);
          _battle(_tokenId, _credBet);
      }
  }
```

- `RapBattle.sol::_battle` **has a weak PRNG, leading to a challenger manipulation of the battle results**
  - **Description:**
    - Weak PRNG due to a modulo on block.timestamp, now or blockhash. These can be influenced by miners to some extent so they should be avoided.
  - **Impact:**
    - A challenger can manipulate the battle result to won the bet.
  - **Proof of Concept:**
    - Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
  - **Recommendation:**
    - Do not use block.timestamp, now or blockhash as a source of randomness

- `RapBattle.sol::_battle` **uses dangerous equality. If the bet is too high and the opponent is too strong, nobody will challenge causing unintended lock to protocol.**
  - **Description:**

- `RapBattle.sol::_battle` requires that both bets must have the same value. The protocol design allows one battle at a time, so if the defender is too skilled and bets too much Cred, nobody will accept the challenge.

- **Impact:**
  - Protocol will locked because the function will be blocked with a giant bet.

- **Proof of Concept:**
  - After a big bet
  - If a user doesn't match the bet, the protocol will be blocked until someone matches

    ▶ Add the code below to `OneShotTest.t.sol`

```
    function testGoOnBlocked() public mintRapper {
        vm.startPrank(address(streets));
        cred.mint(user, 100);
        vm.stopPrank();

        vm.warp(51684120);
        vm.startPrank(user);
        oneShot.approve(address(rapBattle), 0);
        cred.approve(address(rapBattle), 100);
        rapBattle.goOnStageOrBattle(0, 100);
        address defender = rapBattle.defender();
        vm.stopPrank();

        vm.startPrank(challenger);
        rapBattle.goOnStageOrBattle(1, 0);
    }
```

- **Recommendation:**
  - Create a counter to control the battles
  - Create a struct to control the line-up
  - Create a mapping to keep track of battles
  - Redesign the `RapBattle::goOnStageOrBattle` function into a `RapBattle::credDefender` & `RapBattle::credChallenger` functions

    ▶ Create the variables in `RapBattle.sol` as follows

```
+    struct Lineup {
+        address defender;
+        address challenger;
+        uint256 defenderNFTId;
+        uint256 bet;
```

```
+    }
+    uint256 battleCounter = 1;
+    mapping(uint256 battleCounter => Lineup) private stageLimits;
+    error RapBattle__NotEnoughCred();
```

▶ Convert the `RapBattle.sol::goOnStageOrBattle` into `RapBattle.sol::credDefender` and `RapBattle.sol::credChallenger` as follows

```
- 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);
-   }
-}


+ function credDefender(uint256 _tokenId, uint256 _credBet)
external {
+       if (credToken.balanceOf(msg.sender) < _credBet){
+               revert RapBattle__NotEnoughCred();
+       }
+       stageLimits[battleCounter] = Lineup ({
+           defender: msg.sender,
+           challenger: address(0),
+           defenderNFTId: _tokenId,
+           bet: _credBet
+       });
+
+       emit OnStage(msg.sender, _tokenId, _credBet);
+
+       battleCounter++;
+
+       oneShotNft.transferFrom(msg.sender, address(this),
_tokenId);
+       credToken.transferFrom(msg.sender, address(this),
_credBet);
+   }
```

```
+ function credChallenger(uint256 _stageId, uint256 _tokenId)
external {
+       if(credToken.balanceOf(msg.sender) <
stageLimits[_stageId].bet || oneShotNft.ownerOf(_tokenId) !=
msg.sender){
+           revert RapBattle__NotEnoughCredOrItsNotTheNFTOwner();
+       }
+
+       stageLimits[_stageId].challenger = msg.sender;
+
+       // credToken.transferFrom(msg.sender, address(this),
_credBet);
+       _battle(_stageId, _tokenId);
+   }
```

▶ Adjust the `RapBattle.sol::_battle` as follow

```
+ function _battle(uint256 _stageId, uint256 _tokenId) internal {
-       address _defender = defender;
-       require(defenderBet == _credBet, "RapBattle: Bet amounts
do not match");
-       uint256 defenderRapperSkill =
getRapperSkill(defenderTokenId);
+       uint256 defenderRapperSkill =
getRapperSkill(stageLimits[_stageId].defenderNFTId);
+       uint256 challengerRapperSkill = getRapperSkill(_tokenId);
+       uint256 totalBattleSkill = defenderRapperSkill +
challengerRapperSkill;
-       uint256 totalPrize = defenderBet + _credBet;
+
+       //Must adjust this too.
        uint256 random =
            uint256(keccak256(abi.encodePacked(block.timestamp,
block.prevrandao, msg.sender))) % totalBattleSkill;

-       defender = address(0);

+       emit Battle(msg.sender, _tokenId, random <=
defenderRapperSkill ? stageLimits[_stageId].defender :
msg.sender);
-       emit Battle(msg.sender, _tokenId, random <
defenderRapperSkill ? _defender : msg.sender);

        if (random <= defenderRapperSkill) {
-           credToken.transfer(_defender, defenderBet);
+           credToken.transfer(stageLimits[_stageId].defender,
stageLimits[_stageId].bet);
-           credToken.transferFrom(msg.sender, _defender,
```

```
_credBet);
+            credToken.transferFrom(msg.sender,
stageLimits[_stageId].defender, stageLimits[_stageId].bet);
        } else {
-            credToken.transfer(msg.sender, _credBet);
+            credToken.transfer(msg.sender,
stageLimits[_stageId].bet);
        }
-       totalPrize = 0;

-       oneShotNft.transferFrom(address(this), _defender,
defenderTokenId);
+       oneShotNft.transferFrom(address(this),
stageLimits[_stageId].defender,
stageLimits[_stageId].defenderNFTId);
    }
```

- `OneShot.sol` **isn't updating the battles won, breaking the protocol functionality**
  - **Description:**
    - After a battle is finished, the winner should receive a status update on battlesWon. However, this never happens.
  - **Impact:**
    - The bonus over battles won is not applied. Breaking the functionality.
  - **Proof of Concept:**

▶ Add the code below to `OneShot.t.sol`

```
    function testBattleWonNeverUpdated() public mintRapper {
        vm.startPrank(user);
        oneShot.approve(address(rapBattle), 0);
        rapBattle.goOnStageOrBattle(0, 0);
        vm.stopPrank();
        vm.startPrank(challenger);
        oneShot.mintRapper();
        oneShot.approve(address(rapBattle), 1);
        rapBattle.goOnStageOrBattle(1, 0);

        IOneShot.RapperStats memory statsDefender =
oneShot.getRapperStats(0);
        IOneShot.RapperStats memory statsChallenger =
oneShot.getRapperStats(1);

        console.log(statsDefender.battlesWon);
        console.log(statsChallenger.battlesWon);
    }
```

- **Recommendation:**

▶ Update the `OneShot.sol::onlyStreetContract` modifier to accept calls from `RapBattle.sol::_battle` and create the helpe function below

```
+   modifier onlyStreetContract() {
+       require(msg.sender == address(_streetsContract) || msg.sender
== address(s_rap), "Not Allowed");
+       _;
+   }

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

+   event OneShot__RapBattleAddressUpdated();

+   function setRapBattleContract(address _rapBattle) external
onlyOwner {
+       s_rap = RapBattle(_rapBattle);

+       emit OneShot__RapBattleAddressUpdated();
+   }
```

▶ Update the `RapBattle.sol::_battle` as follow

```
    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) {
+           IOneShot.RapperStats memory stats =
oneShotNft.getRapperStats(defenderTokenId);
```

```
+           stats.battlesWon++;
+           oneShotNft.updateRapperStats(defenderTokenId,
stats.weakKnees, stats.heavyArms, stats.spaghettiSweater,
stats.calmAndReady, stats.battlesWon);
            // We give them the money the defender deposited, and the
challenger's bet
            credToken.transfer(_defender, defenderBet);
            credToken.transferFrom(msg.sender, _defender, _credBet);
        } else {
+           IOneShot.RapperStats memory stats =
oneShotNft.getRapperStats(_tokenId);
+           stats.battlesWon++;
+           oneShotNft.updateRapperStats(_tokenId, stats.weakKnees,
stats.heavyArms, stats.spaghettiSweater, stats.calmAndReady,
stats.battlesWon);
            // 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);
    }
```

- `Streets.sol::stake` **function allows users to stake after unstake, breaking the protocol rewards system**
  - **Description:**
    - The protocol documentation states "Staked Rapper NFTs will earn 1 Cred ERC20/day staked up to 4 maximum". However, after a user unstake the NFT, he can stake again breaking the maximum reward established for staked NFTs.
  - **Impact:**
    - A user can take advantage of this vulnerability and collect rewards consistently.
  - **Proof of Concept:**

▶ Add the code below in `OneShotTest.t.sol`

```
    function testIfAUserCanReStakeTheNFT() public mintRapper{
        vm.startPrank(user);
        oneShot.approve(address(streets), 0);
        streets.stake(0);
        assert(
            streets.onERC721Received(address(0), user, 0, "")
                ==
bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))
        );
```

```
        uint256 userBalanceBeforeUnstake = cred.balanceOf(user);

        console.log(userBalanceBeforeUnstake);

        vm.warp((4* 1 days) + 1);

        streets.unstake(0);

        uint256 userBalancesAfterUnstake = cred.balanceOf(user);

        assertEq(userBalancesAfterUnstake, 4);

        oneShot.approve(address(streets), 0);
        streets.stake(0);
        assert(
            streets.onERC721Received(address(0), user, 0, "")
                ==
bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))
        );

        vm.warp((8*1 days) + 1);

        streets.unstake(0);

        uint256 userBalanceAfterSecondStake = cred.balanceOf(user);

        assertEq(userBalanceAfterSecondStake, 8);
    }
```

- **Recommendation:**

▶ Add the following changes into `Streets.sol::stake`

```
+   error Streets__StakingRewardsAlreadyCollected();
    function stake(uint256 tokenId) external {
+       if(stakes[tokenId].startTime != 0){
+           revert Streets__StakingRewardsAlreadyCollected();
+       }

        stakes[tokenId] = Stake(block.timestamp, msg.sender);
        emit Staked(msg.sender, tokenId, block.timestamp);
        oneShotContract.transferFrom(msg.sender, address(this),
tokenId);
    }
```

▶ Add the following changes into `Streets.sol::unstake`

```
    function unstake(uint256 tokenId) external {
        require(stakes[tokenId].owner == msg.sender, "Not the token
owner");
        uint256 stakedDuration = block.timestamp -
stakes[tokenId].startTime;
        uint256 daysStaked = stakedDuration / 1 days;

        // Assuming RapBattle contract has a function to update
metadata properties
        IOneShot.RapperStats memory stakedRapperStats =
oneShotContract.getRapperStats(tokenId);

        emit Unstaked(msg.sender, tokenId, stakedDuration);
-        delete stakes[tokenId]; // Clear staking info

        // Apply changes based on the days staked
        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);
        }

        // Only call the update function if the token was staked for at
least one day
        if (daysStaked >= 1) {
            oneShotContract.updateRapperStats(
                tokenId,
                stakedRapperStats.weakKnees,
                stakedRapperStats.heavyArms,
                stakedRapperStats.spaghettiSweater,
                stakedRapperStats.calmAndReady,
                stakedRapperStats.battlesWon
            );
        }

        // Continue with unstaking logic (e.g., transferring the token
back to the owner)
        oneShotContract.transferFrom(address(this), msg.sender,
tokenId);
    }
```

Medium Severity Vulnerabilities

- **Centralization Risk for trusted owners**
  - **Description:**
    - Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.
  - **Impact:**
    - Contract owner can change the `Streets.sol` address to an arbitrary address and cut the mint of the Cred for staked Rappers,
  - **Proof of Concept:**
    - Access `CredToken.sol::setStreetsContract`;
    - Input an arbitrary address or even the address(0);
    - Done.

- **Missing events access control, leading to difficulty in tracking changes in `OneShot.sol`**
  - **Description:**
    - Detect missing events for critical access control parameters
  - **Impact:**
    - A malicious user can take control of the SmartContracs and update the addresses unnoticed
  - **Proof of Concept:**

    ▶ `OneShot.sol::setStreetsContract`

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

Low Severity Vulnerabilities

- **Missing `address(0)` validation**
  - **Description:**
    - Detect missing zero address validation.
  - **Impact:**
    - Owner can mistakenly add an address(0);
  - **Recommendation:**

    ▶ `CredToken.sol`

```
    function setStreetsContract(address streetsContract) public
onlyOwner {
+       if(streetsContract == address(0)){
+         revert
+       }
        _streetsContract = Streets(streetsContract);
    }
```

▶ `OneShot.sol`

```
    function setStreetsContract(address streetsContract) public
onlyOwner {
+       if(streetsContract == address(0)){
+         revert
+       }
        _streetsContract = Streets(streetsContract);
    }
```

Gas Recommendations

- **Functions not used internally could be marked external**

▶ Found in `CredToken.sol`

```
function setStreetsContract
function mint
```

▶ Found in `OneShot.sol`

```
function setStreetsContract

function mintRapper

function updateRapperStats

function getRapperStats

function getNextTokenId
```

- **Functions only used by the contract could be marked private**

▶ Found in `OneShot.sol`

```
    - `OneShot.sol::getRapperStats` is the rapperStats getter.
-   mapping(uint256 => RapperStats) public rapperStats;
+   mapping(uint256 => RapperStats) private rapperStats;
```

▶ `RapBattle.sol`

```
-   function getRapperSkill(uint256 _tokenId) public view returns (uint256
finalSkill) {}
+   function getRapperSkill(uint256 _tokenId) private view returns (uint256
finalSkill) {}
```

- **State variables that could be declared immutable**

▶ `RapBattle.sol`

```
-   IOneShot public oneShotNft;
+   IOneShot immutable oneShotNft;
-   ICredToken public credToken;
+   ICredToken immutable credToken;
```

- **Unused import**

▶ `OneShot.sol`

```
- import {Credibility} from "./CredToken.sol";
```

Informational Recommendations

- **Naming best practices for imports aren't been followed**

▶ Found in `CredToken.sol`

```
- import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
+ import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
- import "@openzeppelin/contracts/access/Ownable.sol";
+ import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
```

▶ Found in `Streets.sol`

```
-   import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
+   import {IERC721Receiver} from
"@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
```