CCTF

CCTF is a Capture the Flag (CTF) game. CTF in computer security is an exercise in which 'flags' are secretly hidden in purposefully-vulnerable programs or websites. Competitors steal flags either from other competitors (attack/defense-style CTFs) or from the organizers (jeopardy-style challenges).

From CCTF volume 9, we have started to use a decentralized system: Solidity smart contract deployed and players send signed messages. Flags became private keys. We replaced CTFd to our smart contract, but it needs improvements.

Address: *0x36a1424da63a50627863d8f65c0669da7347814a*

The main problem is that all the transactions on the blockchain are visible to everyone.

Concept about the vulnerability

Scan the address using polygonscan.com

https://polygonscan.com/address/0x36a1424da63a50627863d8f65c0669da7347814a

Get the source code of the smart contract

You can use http://remix.ethereum.org/ to play around with it.

```
/**
 *Submitted for verification at polygonscan.com on 2022-08-24
*/

// SPDX-License-Identifier: Apache-2.0
// Authors: six and Silur
pragma solidity ^0.8.16;

contract CCTF9 {
  address public admin;
  uint256 public volStart;
  uint256 public volMaxPoints;
  uint256 public powDiff;
  bool public started;

  enum PlayerStatus {
    Unverified,
    Verified,
    Banned
}

struct Player {
  PlayerStatus status;
  uint256 points;
}

modifier onlyAdmin {
  require(msg.sender == admin, "Not admin");_;
}

modifier onlyActive {
  require(started == true, "CCTF not started.");_;
}

struct Flag {
  address signer;
  bool onlyFirstSolver;
  uint256 points;
  string ski_name;
}

mapping (address => Player) public players;
mapping (uint256 => Flag) public flags;

event CCTFStarted(uint256 timestamp);
event FlagAdded(uint256 indexed flagId, address flagSigner);
event FlagRemoved(uint256 indexed flagId);
event FlagSolved(uint256 indexed flagId, address indexed solver);

constructor(uint25_volMaxPoints, uint25_powDiff) {
  admin = msg.sender;
  volMaxPoints _volMaxPoints;
  powDiff _powDiff;
  started = false;
}

function setAdmin(addres_admin) external onlyAdmin {
  requir_admin != address(0));
  admin _admin;
}

function setCCTFStatus(boo_started) external onlyAdmin {
  started _started;
}
```

```
  function setFlag(uint25_flagId, addres_flagSigner, boo_onlyFirstSolver, uint25_points, st
    flag_flagId] = Fla_flagSigner_onlyFirstSolver_points_skill);
    emit FlagAdde_flagId_flagSigner);
  }

  function setPowDiff(uint25_powDiff) external onlyAdmin {
    powDiff _powDiff;
  }


  function register(string memor_RTFM) external {
    require(players[msg.sender].status == PlayerStatus.Unverified, 'Already registered or ba
    //uint256 pow = uint256(keccak256(abi.encodePacked("CCTF", msg.sender,"registration", no
    //require(pow < powDiff, "invalid pow");
    require(keccak256(abi.encodePacked(_re_it')) == keccak256(abi.encodePacke_RTFM)));  // R
    players[msg.sender].status = PlayerStatus.Verified;
  }

  function setPlayerStatus(address player, PlayerStatus status) external onlyAdmin {
    players[player].status = status;
  }


////////// Submit flags
    mapping(bytes32 => bool) usedNs;                       // Against replay attack (we only
    mapping (address => mapping (uint256 => bool)) Solves;     // address -> challenge ID ->
    uint256 public submissi_succe_count = 0;               // For statistics

    function SubmitFlag(bytes3_message, bytes memory signature, uint25_submitFor) external o
        require(players[msg.sender].status == PlayerStatus.Verified, "You are not even playi
        require(bytes3_message).length <= 256, "Too long message.");
        require(!usedN_message]);
        usedN_message] = true;
        require(recoverSigne_message, signature) == flag_submitFor].signer, "Not signed with
        require(Solves[msg.sender_submitFor] == false);

        Solves[msg.sender_submitFor] = true;
        players[msg.sender].points += flag_submitFor].points;
        players[msg.sender].points = players[msg.sender].points < volMaxPoints ? players[msg

        if (flag_submitFor].onlyFirstSolver) {
            flag_submitFor].points = 0;
        }

        submissi_succe_count = submissi_succe_count + 1;
        emit FlagSolve_submitFor, msg.sender);
    }

    function recoverSigner(bytes3_ethSignedMessageHash, bytes memor_signature) public pure r
        (bytes32 r, bytes32 s, uint8 v) = splitSignatur_signature);
        return ecrecove_ethSignedMessageHash, v, r, s);
    }

    function splitSignature(bytes memory sig) public pure returns (bytes32 r, bytes32 s, uin
        require(sig.length == 65, "Invalid signature length");
        assembly {
            r := mload(add(sig, 32))
            s := mload(add(sig, 64))
            v := byte(0, mload(add(sig, 96)))
        }
    }

////////// Check status, scores, etc
  function getPlayerStatus(addres_player) external view returns (PlayerStatus) {
    return player_player].status;
  }

  function getPlayerPoints(addres_player) external view returns (uint256) {
    return player_player].points < volMaxPoints ? player_player].points : volMaxPoints;
  }

  function getSuccessfulSubmissionCount() external view returns (uint256){
      return submissi_succe_count;
  }
}
```

Looking at the transaction

There are 3 frequently used methods in the history of the transactions.

Set Flag

Admin users can put flags to store. This is not vulnerable.

Name	Type
_flagId	uint256
_flagSigner	address
_onlyFirstSolver	bool
_points	uint256
_skill	string

Register

Users can register using the `I_read_it` string.

Name	Type	Data
_RTFM	string	I_read_it

Submit Flag

After registering, users can submit flags.

Name	Type
_message	bytes32
signature	bytes
_submitFor	uint256

Stealing flags from other users

Since everything is visible, users can see each other's submited flags. Reading the source code of the smart contract it reveals how the flags are validated. It checks the signer of the message and the signature of the request, and compares with the flag's signer. Because of all of these informatoins are publicly accessable, it is posible to retrieve the flags based on other user's submisions.

Concept about fixing this flag stealing vulnerability.

A noninteractive Zero-knowledge proof

A noninteractive ZKP, where two participants have a shared secret key, which is enough to prove that one of them knows some secret information without revealing the information itself.

The user makes a proof about having the flag, while the smart contract can validate that the user has it, without actually revealing the content of the flag.