Dapp in Ethereum – Report

Master's degree in Cybersecurity – University of Pisa
Peer to Peer & Blockchain

Andrea Giuliani

Contents

1) INTRODUCTION

The final project of “P2P and Blockchain” course has the aim to develop a Dapp, implementing a Battleship game with the Ethereum blockchain. There are three main roles during a game that must be taken under consideration:

- One player in the role of game creator.
- One player in the role of joiner of a previously created game.
- Smart contract deployed in the Ethereum blockchain in the role of game controller.

1.2) GAME FEATURES

The full project was developed thinking to obtain a secure and safe back-end, and a primitive front-end to call functions in the smart contract and evaluate its functionality. It gives the possibility to create a game, join a game using a unique ID or randomly, propose new deposit, chose board size and amount of deposit, delete a game by creator, play the battle, accuse of inactivity, and obtain the final payment from the smart contract.

This game provides three different configurations of board and for each board a fixed number of ships located by players in the strategy phase. *(2x2: 2 ship unit, 4x4: 6 ship units, 8x8: 23 ship units; Ship unit is equivalent to a single board cell).*

With the use of deposit in ETHs and implemented checks, a player may have a cryptocurrency loss in case of cheating, encouraging regular game. Deposit is intended as bail returned at the end of the game only in case of good behavior of player. In the opposite case the deposit will be sent to the opposer.

1.3) DIFFERENCE WITH TRADITIONAL GAME

The main project simplification regards constraints on ships length. To obtain a more user-friendly front-end it was implemented a touchpad or mouse interface to place ships and shot torpedoes without taking into consideration the possibility for a player to not follow rules and put all ships of one ship unit. I did not consider also the “sunk” response because absence of different length ship constraints.

2) BACK-END

It is composed of a single Solidity file, the Ganache blockchain and Node.js. Some provided functionalities are storing all information about the game state (23 variables in a storage structure), defining constraints for users, checking all tainted input provided from the front-end and checking response validity with Merkle Tree data structure.

Unique ID → unique identifier of the game with ***uint8*** type. Implemented with a counter. The overflow problem is managed reactivating old ***gameId***, if games are ended or deleted. There is also a required statement for not more than 256 ***gameId*** at the same time. Of course, this is a theoretical number, and I did not provide any tests to guarantee 256 games concurrently worked.

Board → it is implemented as an array with 0 or 1 value with length equal to the number of cells in board. At each shot, smart contract stores these values if there is a ship or not in the target cell.

Compilation → compilation is done with compiler version 0.8.13, but it is compilable with versions between 0.4.22 and 0.9.0. Because smart contract size I use an optimization provided by truffle with "2000" runs, limiting the number of calls to OPCODE in smart contract life.

- **Creation, Delete Game**

There must be a correct size of board and the transaction to the smart contract must have an ETH value equal to the deposit previously selected. Only the creator can delete the game and only if there is not already a player interested in it.

- **Propose Deposit**

Players can propose a new deposit to the creator. If he/she denies there are no updates in the game state. In other case only the creator can change deposit with the ***changeDeposit()*** function in two different mechanisms, accepting the propose:

- If the new amount is higher than the old deposit, the creator can send a transaction with a number of ETHs equal to the difference between propose and old deposit.
- If the new amount is lower than the old deposit, creator receives a payment from the smart contract of the amount difference between propose and old deposit.

The proposed phase can go forward until a ***changeDeposit()*** and a ***payDeposit()*** functions are called consecutively. After that, automatically game starts.

- **Merkle Proof:**

Merkle proof is used in two different situations. After each shot torpedo, the target player must submit the result "hit" or "miss" with respective Merkle proof for that board cell.

After the end of game if there were not cheaters during the battle, I implemented two checks:

1) The winner must submit an array with his/her ships, an array with salt of ships randomly chosen in the board creation and an array with Merkle proof for each ship position. Smart contract computes the hash value with ***keccak256()*** function and recompute the Merkle root for all ships.
2) Smart contract checks also that all ships were located on board by the winner.

Merkle root is submitted by two players after the ship placement phase.

```
For(uint8 j = 0; j < values.length; j++) {

            uint256 tmp = values[j] + seeds[j];
            computed_hash = keccak256(abi.encode(tmp));

            for(uint8 I = j*size_merkle_proof; I < (j+1)*size_merkle_proof; i++)
{

                if (indexes[j] % 2 == 0) {
                    computed_hash = keccak256(abi.encodePacked(computed_hash,
merkle_proofs[i]));
                } else {
                    computed_hash = keccak256(abi.encodePacked(merkle_proofs[i],
computed_hash));
                }
                indexes[j] = indexes[j] / 2;
            }

            if(computed_hash != merkle_root_ver) {
                games[gameId].cheat = true;
                games[gameId].winner = opposer;
            }
        }
```

- **Accuse and End Game**

When player wants, he/she can accuse the opposer to inactivity and at that time smart contract stores a time limit in which the accused must make a move otherwise game ends and all deposit is sent to the accuser. The end game verification is done through checking ***remaining_ship_counter*** for each player. Finally, there is a payment transaction from smart contract to one or all players depending on context. If a player has cheated, all deposit is sent to the opposer but in the opposite case deposit is split to both players.

3) FRONT-END

It is implemented with vanilla JavaScript, HTML and CSS files with other plugins used as bootstrap, web3 and web3Utils. The user interface is quite simple and intuitive with different html ***divs***, showed or hidden by user click on buttons. All files about front-end are organized in *src* directory in this way:

- *css* (dir): files for html style in CSS language exploiting bootstrap plugin.
- *images* (dir): all images used in the application.
- *js* (dir): containing ***app.js*** file.
- ***Index.html***

- **Events handle:**

I defined several events on buttons to execute JavaScript code following user actions and another function to manage events emitted by the smart contract, to update the user-side depending on data returned by back-end.

```
 bindEvents: async function () {
    $(document).on("click", "#createNewGame-btn", App.newGameView);
    $(document).on("click", "#createGame-btn", App.createGame);
    $(document).on("click", "#findGameRandom-btn", App.findGameRandom);
    $(document).on("click", "#deleteGame-btn", App.deleteGame);
    $(document).on("click", "#backToMainMenu-btn", App.backToMainMenu);
    $(document).on("click", "#findIdGame-btn", App.findIdGame);
    $(document).on("click", "#payDeposit-btn", App.payDeposit);
    $(document).on("click", "#submitPropose-btn", App.submitPropose);
    $(document).on("click", "#acceptPropose-btn", App.acceptPropose);
    $(document).on("click", "#submitAccuse-btn", App.submitAccuse);
    $(document).on("click", "#verifyAccuse-btn", App.verifyAccuse);

    console.log("Button listeners loaded!");

  },
```

```
if (events.event == “PaidDeposit” && events.args.tx_sender !=
web3.eth.defaultAccount) {
          document.getElementById(“waitingRoomCreator”).style.display = “none”;
          document.getElementById(“shipPlacement”).style.display = “block”;

          App.configureBoard(board_size);
```

- **Board creation:**

I implemented a board, defining for each board cell a new html *div* inside a grid element adding an ***eventListener()*** on click, to give the possibility to put ships using mouse or touchpad. The same idea was for ***shotTorpedo()*** function but in a second grid, represented the opposer board.

```
for (let i = 0; i < size; i++) {
      for (let j = 0; j < size; j++) {
        const cl = document.createElement("div");
        cl.classList.add(my_cell_name);
        cl.dataset.row = i;
        cl.dataset.col = j;
        cl.addEventListener("click", (location) => App.shipPlacement(location));
        my_matrix.appendChild(cl);
      }
    }
```

- **Merkle Tree:**

The front-end crafts a Merkle Tree creating hashes as sum of a random salt (1 digit number) and value 0 or 1 in absence or presence of a ship in the board. I used web3Utils plugin calling ***soliditySha3()*** function to create 256 bits hash leaves. This first array of hashes is stored in the first position of array ***merkle_tree***, and iteratively all other levels are created in the same way taking under consideration hashes of lower-level tree.

```
var seed = Math.floor(Math.random() * 10);
        seed_matrix[i][j] = seed;

        var tmp = my_board[i][j] + seed;

        console.log(tmp);
        temp.push(window.web3Utils.soliditySha3(tmp));
```

```
const left_child = temp[j];
     const right_child = temp[j + 1];

     next.push(window.web3Utils.soliditySha3(left_child + right_child.slice(2)));
```

4) USER MANUAL

To try application is necessary Ganache, Truffle and MetaMask. First, it is necessary to execute Ganache and run the Ethereum blockchain locally in a properly configured workspace. It is also necessary to prepare different accounts in MetaMask importing private keys from Ganache accounts. Contract must be compiled, going to contract fold and execute "**truffle migrate**". Some useful information about deployment is displayed as cost, gas used and others.

Next step is to execute a lite-server session and run front-end code with "**npm run dev**" command in the main project directory. If we want to open two sessions on the same network, it is necessary to disable synchronization settings, going to *"localhost:3001"*. Now you can open two browser pages, go to *"localhost:3000"* and with two different MetaMask accounts evaluate the application.

If you do not want to change at each transaction of different players the MetaMask account, you can execute in two different command prompt "npm run dev", always in the main project folder. Then open two different browsers with MetaMask in *"localhost:3000"* and *"localhost:3002"*. In this way you can switch browsers without changing MetaMask for different player transactions.

5) SMART CONTRACT VULNERABILITIES

- **UNDER/OVERFLOW**: From compiler version 0.8 there is under/overflow check at language level. So, in case of compiling with lower version there could be no guarantees that some underflow or overflow may not occur, because ***SafeMath*** library is not explicitly imported and used.

- **DoS**: There is no check on DoS attack limiting the number of games that one single user can create. If the creator starts 256 games, service will be not available for other players.

- **GAS LIMIT**: Some operations could consume too much gas and produce out-of-gas errors with revert in transactions. For example, when we iterate through arrays this error can appear. One solution could be using more sophisticated data structure than arrays.

- **MINERS**: In opposer accusation I consider blockchain state to set the time limit and attacker can tries to manipulate blockchain and produce a functional error in my application. For all those attacks regarding miners, there is no protection.

6) EVALUATION OF GAS

Gas cost evaluation for each contract function, considering one single function call without complex tests on gas consumed:

- createGame → 300k
- deleteGame → 60k
- joinGameRandom → 46k
- joinGameId → 40k
- proposeDeposit → 54k
- changeDeposit → 55k
- payDeposit → 56k
- submitMerkleRoot → 60k
- shotTorpedo → 40k
- shotResult → 61k
- submitAccuse → 55k
- verifyAccuse → 36k
- removeAccuse → 33k
- verifyEndGame → 52k (with 2x2 board, it will be higher in 8x8 due to 23 Merkle root to check)
- endGameCheater → 30k
- transactionEndGame → 53k

Gas cost for a game with 8x8 board and only one ship, missing all ships and accepting the first creator's deposit:

- **Creator transactions: 1 x createGame, 1 x submitMerkleRoot, 64 x shotTorpedo , 63 x shotResult, 1 x verifyEndGame, 1 x transactionEndGame . [6.86 million]**

- **Player transactions: 1 x joinGameId, 1 x payDeposit, 1 x submitMerkleRoot, 63 x shotTorpedo, 64 x shotResult. [6.58 million]**

The ***verifyEndGame()*** function is the only function in smart contract with high gas cost variance depending on selected board. My evaluation does not consider a real end game with 23 ships as in 8x8 board. My total gas cost evaluation has a remarkably high approximation.

Total gas cost for this game: 13.5 million