University of Pisa

Department of Information Engineering

Master's Degree Course in Cybersecurity

Project Report:

P2P System and Blockchain

Prof. Laura Emilia Ricci

Made by:

Alessandro Niccolini

Academic Year 2022-2023

Contents

Chapter 1

Introduction

This project focuses on a Battleship game, a traditional game in which two players position a fixed number of ships and after that phase try to find the other player's ships. Ship placement and tactical engagement are the two separate phases of the game, which are defined by a secret battlefield and hidden information. On centralized systems, it has traditionally found its digital manifestation. The Ethereum blockchain, a platform renowned for its inherent benefits including tamper-resistance, transparent rule enforcement via smart contracts, secure and instant reward distribution, and a spirit of equality that transcends censorship or restriction, is where this project seeks to re-imagine the Battleship experience.

The Battleship experience is transferred to the Ethereum blockchain, where a wide range of options are made possible. Ship placements will be protected after they are established thanks to the blockchain, which is famed for its immutable ledger. Additionally, it takes on the responsibility of validating ship positions, preventing fraud or the misrepresentation of results, facilitating secure reward systems through the use of cryptocurrency or tokens, and enforcing penalties for unfair play or inaction, temporarily freezing deposited assets within smart contracts.

Further, the characteristics of the blockchain enable unlimited participation, this in particular is possible due to the permission-less structure and this also eliminates the possibility of censorship or exclusion. With the use of that uniqueness, the project can lead to a very functional digital version of the famous board game. Help cover many topics, from the usage of Merkle trees to ensure ship placements up to the organization of torpedo launches and the determination of the winners.

Figure 1.1

Chapter 2

Implementation choices

The structs used are in an interface called *IntBattleshipStruct.sol*, inside there are some enum that help the construction of complex structures. The first relevant struct is the *ShipPosition*, how it's shown inside the code 2.1. The ship is composed of a length, the axis and a direction (in which the ship grows).

```
1  struct ShipPosition {
2      uint8 shipLength;
3      uint8 axisX;
4      uint8 axisY;
5      ShipDirection direction;
6  }
```

Listing 2.1: ShipPosition struct

The *BattleModel* struct contains the main information, that allows the player to store the info of the battle.

```
1  struct BattleModel {
2      uint256 stake; // Ethers was staked for this battle
3      address host; // Address of the host player
4      address client; // Address of the client connected
5      address turn; // Address indicating whose turn it is to play next
6      bool isCompleted; // Battle has been completed
7      bool opponentStakeRefundable; // Mark the opponent's stake as refundable
8      bool clientStakeFrozen; // The client has expired his time to play
9      bool hostStakeFrozen; // The host has expired his time to play
10     address winner; // Address of the winning player;
11     GamePhase gamePhase; // The game phase
12     uint256 maxTimeForPlayerDelay; // If a player does not play after this time elapses,
13     // then the contract will freeze the stake value of the player
14     // and the other will obtain the amount
15     uint256 createdAt; // Time Created
16 }
```

Listing 2.2: BattleModel struct

The *PlayerModel* struct saves the index of the ship correlated with the leaves to perform the correct check in the attack phase, but without exposing the directly to the player the ship position.

```
1  struct PlayerModel {
2      ShipPosition[] shipPositions;
3      // Array of ship positions
4      uint8[] leafIndexX; // For each shipPosition I save the leaf index X
5      uint8[] leafIndexY; // For each shipPosition I save the leaf index Y
6      uint8[] leafIndexShipPosition;
7      // correspond to the shipPosition relative
8      // to the leafIndexX[i] and leafIndexY[i]
9      bytes32[][] leaves; // Array of Merkle tree leaves
10 }
```

Listing 2.3: PlayerModel struct

It also implemented a support struct called *GamePhaseDetail* to help track the stake value, the penalty amount the game phase in which the player is, and the max time for the players to play, before freezing the amount of stake committed by the player and consider

that the player has left the game.
Another relevant struct is the *LobbyModel*, which saved the information for the lobby, such as if there is a slot free or not, the address of the address of player that has joined, and the two Merkle root hash of the player.

The game board size allowed is 4×4 or 8×8. While the number of ships is equal to the game board size divided by 2.

The project uses two main contracts, the Battleship.sol in which implemented the function that allows the user to perform the main actions:

- **createLobby**, that allows the user to create a new game, preparing the lobby struct with the new information. And waiting for a player to join.
- **joinLobby**, that lets the player join an existing game created con the previously mentioned function. Exploit the creator address player to join. In this function, the lobby struct is filled with missing information. When we call this function we have already set the ship positions and created the Merkle Root tree, so we start to create the battle struct and set the useful information like the Merkle root, the next player turn, and the battle ID that represents this specific battle.
- **attack**, that lets the player perform the attack using the attacking position coordinates and the generated proof of the leaf to allow the verification check later to see if the player hasn't cheated about the ship position. Inside this function we check also if the time has elapsed to verify that the players are still present and active during the game, otherwise, a penalty is applied. After updating all the intermediate information like the position attacked by the player we emit the events that indicate the shot status and the attack. And finally, check if there is a winner.

Inside this contract, are presents other functions like *freezeDeposit* (to freeze the deposit and apply a penalty in case of cheat or players that don't perform any actions), *checkForWinner* (perform the right check with ship positions and decide if a player win, performing the reward) and others small function to perform the transfer of stake.

The second main contract is the *BattleshipStorage* contract which stores the main variables and information that the Battleship contract uses to achieve and player the game in a correct way. The game logic is inside this contract; are presents functions to create the leaves, to create the Merkle proof, to create the Merkle root, to verify the proof, to verify that a ship has been Hit or not, to set the initial ship positions, to update the various struct saved and so on. A strong set of checks and controls has been carefully built into the execution of this contract to meet the crucial requirements explained in the project description.
The contract analyzes player actions to prevent cheating on torpedo shot outcomes. It seamlessly incorporates a reward system for winners, allowing for the secure and timely distribution of tokens or cryptocurrency from the smart contract's balance. Finally, it uses a penalty mechanism to discourage cheating players, who will be punished by freezing their deposited funds. Together, these safety measures help maintain trust, equity, and integrity in the blockchain-based Battleship game.

Chapter 3

Guide Demo

For the concern about the game execution is possible to use the combination of *Truffle* with *Ganache* to test the game and check that the game properties are satisfied. After the right settings for the truffle-config.js file, is possible to use the *$ truffle migrate* command, which allows to streamlines the development and deployment of the smart contracts by making it easier to deploy. It works well with the local blockchain Ganache for quick testing and deployment. Without paying real gas fees or waiting for block confirmations on the Ethereum mainnet or testnets, Ganache offers a convenient and secure environment for testing your contracts.

For the execution of all the tests is possible to use the command *$ truffle test –compile-none*. Instead for the execution of a single test is possible to specify the name of the test in this way: *$ truffle test .\test\BattleshipTest.js ––compile-none*, in this specific case is test an execution in which a game between two players is set and we have that one of them win. During the game, the main phase are:

1. Set ship positions by using *setShipPositions*, this function acts as a crucial building component for the game's startup and ship placement. The player's address and an array of ship lengths, X and Y coordinates, and ship directions are the inputs. Through some checks, control that the lengths of these input arrays match before continuing to ensure consistency. The integrity of the game is preserved by an anti-cheat check that stops a player from putting more ships than permitted. After a ship is placed, the method calls an internal function called transformShipPosition to turn the ship locations into a Boolean matrix. This function manages important indices for tracking and verification purposes as well as ensuring that ships stay within the game board's limits. Additionally, it makes sure that placed ships' total lengths follow established guidelines. Finally, depending on these ship positions, Merkle tree leaves are produced, which contributes to the cryptographic proof process for the game's ship position verification. An essential part of the Battleship smart contract, setShipPositions enforces fair play and anti-cheating rules in addition to initializing the game state. As an implementation choice, the ship's number is set inside the contract (2) for the respective board size used (4×4). The ship lengths should start from 1 and increase by one up to the maximum number of ships set.

2. Generate Merkle root hash, by using the leaves calculated inside the setShipPositions.

3. Create the lobby, the first player creates the lobby by using the function *createLobby*, the player decides the value to use as stake and emits the event *PlayerCreatedLobby*.

4. Join the lobby, the second player, knowing the player One address can join the lobby with the function *joinLobby*. Inside it the main information is updated and the battleId of the game is created. At this point, the join player must start the attack.

5. Game representation, instead of using a front-end for the graphical representation, it is used a text-based representation of

the board. That uses some function declared inside the test file. The sea is represented by 🌊, the hit by 💥 and the miss by 💦.

6. Set input for the attack function, in which are set the attack position and the proofLeaf, this last is generated by using the function *generateProof* provided by the BattleshipStorage contract.

7. Attack the adversary board, by using the *attack* function provided by the Battleship contract. The result can be viewed by analyzing the event emitted, such as *ConfirmShotStatus* and *AttackLaunched*. The players continue those attacks up to the moment in which all the ships are sunk. Every time that the attack function is called a different player must have called it.

8. Declare a winner, at some point, the *checkForWinner* function inside the attack, will see that a player has hit all the right positions. After all the anti-cheat and correctness check a winner will be declared and the event WinnerDetected will be viewed by the players.

Figure 3.1a, shows the 3rd attack of player One and figure 3.1b shows the 4th attack of player Two.

(a)

(b)

Figure 3.1

Chapter 4

Evalutation of the gas cost

Analyzing the results of the *deploy_battleship.js* migration script will allow us to determine how much gas is used by the smart contract's functions. BattleshipStorage and Battleship are the two contracts that are deployed by the deploy_battleship.js script. The Battleship contract is deployed second after the BattleshipStorage contract. The result of the deployment script reveals that while the BattleshipStorage contract required the deployment of 4089430 gas units, the Battleship contract required the deployment of 2650566 gas units. The two contracts will cost 6740006 gas units to deploy altogether. The total cost of deploying both contracts is 0.016849990047179972 ETH at a gas price of 2.5 gwei. Basically, this is the price of implementing the contracts. Depending on the function being performed and the complex nature of the process, different fees may be associated with using the services provided by the contracts. To convert the cost of the gas from ETH to EUR, we can use the following formula:

$$Gas_{EUR} = Gas_{ETH} \times ETH_{EUR} \tag{4.1}$$

At the time of writing, the ETH price in EUR is approximately $1,560$ EUR. Therefore, the cost of deploying the two contracts in EUR is:

$$\begin{aligned} Gas_{EUR} &= 0.0168\ ETH \times 1,560\ EUR/ETH \\ &= 26.21\ EUR \end{aligned}$$

For which concern the BattleshipTest.js, a board 4×4 is used and player Two wins in very few attacks, the evaluation is the following:

With $6,718,946$ gas units, the test's overall gas cost is quite small. The *attack()* method, which averages $220,105$ gas units in cost, is the most expensive one in the test. The gas cost of the attack() method might vary based on the size of the game board and the complexity of the attack, so it's crucial to keep that in mind. Using the previous formula and just multiplying for the gas units, the cost of the BattleshipTest.js test file in EUR is:

$$\begin{aligned} Gas_{EUR} &= 0.0000025\ ETH \times 1,560\ EUR/ETH \\ &\quad \times 6,718,946 gas_{units} = 26,204\ EUR \end{aligned}$$

Instead for a board size of 8×8, in which each player misses all the guesses, the overall cost of the game is relatively high, at $45,825,612$ gas units. This is because the *attack()* function is called 127 times, and the attack() function is relatively expensive. Therefore, the cost of the game in EUR is:

$$\begin{aligned} Gas_{EUR} &= 0.0000025\ ETH \times 1,560\ EUR/ETH \\ &\quad \times 45,825,612 gas_{units} = 178,720\ EUR \end{aligned}$$

Chapter 5

Vulnerabilities Analysis and Conclusion

As shown in the GitHub site, *Slither is a Solidity & Vyper static analysis framework written in Python3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.*
Based on the results of the *Slither* static analysis tool, the following is an examination of the potential weaknesses in the Battleship contracts:

- **Vulnerability:** Ether can be sent to any address using the attack() method. This is a potentially harmful flaw because it could let an attacker steal ether from the contract.
  The vulnerability has been fixed by including a check to make sure the recipient is a trusted user.

- **Vulnerability:** The setShipPositions() function is vulnerable because it does not check the input information. As a result, an attacker might be able to set up fraudulent ship placements and compromise the game.
  The vulnerability has been fixed by analyzing the input data to make sure that the ship locations are appropriate.

The Slither analysis program has discovered a number of additional potential vulnerabilities in addition to the ones mentioned above, including:

- State variables not in use;
- Unused features;
- Missing statements requiring;
- Incorrect application of modifiers;

5.1 Conclusion

In the end, this project shifted the traditional game of Battleship to the Ethereum blockchain by using its unique features like transparency, tamper resistance, and the ability to use smart contracts for establishing rules. Two key contracts, 'BattleshipStorage' and 'Battleship,' which manage critical game elements, from ship deployment to attack phases, are successfully implemented in the project. These agreements guarantee that ship positions are legitimately established, preventing fraud and misrepresentation. The blockchain also makes it possible to maintain fairness by using secure systems for player rewards and penalties. Additionally, censorship is avoided thanks to the permissionless structure of the blockchain. While ensuring confidentiality and integrity, the project uses Merkle trees for ship placement verification. The examination of gas prices shows that the contracts are not very affordable, but in the future works some improvements of the contracts are possible. Otherwise, one possibility is to develop new layer-2 solutions that are more efficient and easier to use. Layer-2 solutions can offer significantly lower gas fees than the Ethereum mainnet.