



## Playing Battleship on Ethereum

---

Project for P2P Systems and Blockchains - UniPi 2022/2023

Antonio Osele

04/09/2023

## Contents

List of Tables . . . . .	2
List of Figures . . . . .	2
<b>1 Design Choices</b>	<b>3</b>
1.1 Backend . . . . .	3
1.1.1 Solidity contracts . . . . .	3
1.1.2 Rules . . . . .	3
1.1.3 Merkle Tree proofs . . . . .	4
1.1.4 Inactive player . . . . .	4
1.1.5 Forfeit . . . . .	4
1.2 Frontend . . . . .	4
1.2.1 Vue.js . . . . .	4
1.2.2 Bootstrap . . . . .	4
1.2.3 Merkle Trees . . . . .	4
<b>2 Vulnerabilities</b>	<b>5</b>
2.1 Delay exploit . . . . .	5
2.2 Cheating . . . . .	5
2.3 Re-entrancy attacks . . . . .	5
2.4 Safe Math . . . . .	5
2.5 Alternative client . . . . .	5
<b>3 Gas evaluation</b>	<b>6</b>
<b>4 User manual</b>	<b>7</b>
4.1 Running instructions . . . . .	7
4.2 Tests . . . . .	7
4.3 Demo . . . . .	7

## List of Tables

1 Gas cost of functions . . . . .	6
-----------------------------------	---

## List of Figures

1 Homepage . . . . .	7
2 Creating a game . . . . .	8
3 Waiting for an opponent . . . . .	8
4 Joining a game . . . . .	9
5 Depositing the bet . . . . .	9
6 Placing ships . . . . .	10
7 Playing the game - Beginning . . . . .	10
8 Playing the game - Shots taken . . . . .	11
9 Winner verification . . . . .	11
10 Withdrawing the prize . . . . .	12

# 1 Design Choices

In this chapter we explain the various design choices that were taken for the creation of this project, along with the technologies used to implement it.

## 1.1 Backend

The project was developed using Node.js 20.2.0, Truffle 5.11.2, Solidity 0.8.19, web3 1.10.0 and Ganache UI 2.7.1.

### 1.1.1 Solidity contracts

The smart contracts were developed with Solidity. It was decided to use two contracts, the first to handle the games, from their creation to their removal, while the second is the one that provides the gameplay functionalities. This was done for a few reasons:

- **Separation of concerns:** by separating the creation and joining of games from the gameplay, the contracts adhere to the principle of separation of concerns. Each contract is responsible for a distinct aspect of the system's functionality, promoting code organization and maintainability.
- **Easier development:** while it was technically feasible to implement all functionalities within a single contract, opting for this approach would have adversely affected code readability and the ability to conduct thorough audits. By segregating the functionalities into distinct contracts, it becomes easier to analyze and reason about and test each contract independently.
- **Modularity and scalability:** by separating the creation and joining of games from the actual gameplay, the system becomes more modular and scalable. It allows for easier maintenance and upgrades in the future. If there is a need to enhance or modify the game-playing logic, it can be done without affecting the contract responsible for game creation and joining.
- **Improved security:** handling multiple games concurrently within a single contract would have increased the likelihood of introducing subtle vulnerabilities that are difficult to detect. For instance, there could be a risk of the contract mistakenly acting upon the wrong game. By employing separate contracts for each game, this potential issue is mitigated.

### 1.1.2 Rules

Since the focus of the project was the computation of the Merkle tree, it was decided to simplify some rules of the game to speed up the development. The size of the board was fixed to be  $8 \times 8$ , with each board having 10 ships of size  $1 \times 1$  that can be placed in any configuration. As such, each cell can either have a ship or not, and this information is hashed with a random salt to prevent the

exposure of the location of the ships. Each cell represents a leaf of the Merkle tree, and the root of such tree is sent to the contract to validate the proofs that the client sends.

### **1.1.3 Merkle Tree proofs**

The MerkleProof OpenZeppelin contract was used to verify the Merkle Tree proofs. This contract is part of a larger library of secure smart contracts and was used to have a proven and efficient foundation to implement the verification functions.

### **1.1.4 Inactive player**

The reporting mechanism for inactive players works by having a "report" button that any player can click to flag the opponent. After clicking it, the game waits for the reported player to make an action within a set time limit of 5 blocks mined from when the report was sent. If the reported player makes a move after this time runs out or if the other player uses the verify function, they will automatically lose the game.

### **1.1.5 Forfeit**

At any point during a game, before the winner has been verified, a player can decide to forfeit. By doing this the opponent will win by default and the game will end.

## **1.2 Frontend**

### **1.2.1 Vue.js**

It was decided to develop the frontend using the Vue.js framework. Vue is an approachable, performant and versatile framework for building web user interfaces.

### **1.2.2 Bootstrap**

Bootstrap is a powerful, extensible, and feature-packed frontend toolkit that was used to give the interface a better look.

### **1.2.3 Merkle Trees**

Another library from OpenZeppelin was used, merkle-tree. It's a library used to generate Merkle Trees and Merkle Proofs, that works in tandem with the MerkleProof contract used in the backend.

## 2 Vulnerabilities

In this chapter we go over the potential vulnerabilities of the project, with the solutions that were applied to patch them where possible.

### 2.1 Delay exploit

The ability to report a player prevents many problems but also introduces the possibility for a malicious player to delay a game. This is done by continuously waiting just below the 5 block limit after being reported and then acting, slowing the game for every action.

### 2.2 Cheating

Since the loser probably didn't cheat, the verification is done only on the board of the winner at the end of the game. This doesn't take into consideration the possibility of both players cheating, since the preferred outcome of the game in that situation is uncertain.

### 2.3 Re-entrancy attacks

The potential vulnerability to re-entrancy attacks is prevented by the implementation of the contracts itself. Since every game has its own contract, the deposit amount doesn't need to be stored anywhere, awarding to the winner the full balance of the game's contract. Also just before the transfer is actually performed, the game phase is updated, as a further security measure.

### 2.4 Safe Math

Arithmetic underflow/overflow attacks aren't possible since, as of Solidity version 0.8, underflow and overflow checks are implemented on the language level. The contracts were developed targeting version 0.8.19, so the use of the Safe-Math library isn't needed.

### 2.5 Alternative client

The client that has been developed incorporates 256-bit cryptographically secure random salts. However, it is important to note that anyone has the ability to create an alternative client. If a malicious client intentionally utilizes weak salts during the creation of the Merkle tree, it could potentially expose vulnerabilities that could be exploited.

### 3 Gas evaluation

A test was used to evaluate the gas cost of the functions. This test runs all the contracts' functions by simulating a real game and outputs the gas used for each in the console. The table below shows the results.

Contract	Function	Gas used
HandleGames	createGame	2,110,228
HandleGames	getRandomGame	55,656
HandleGames	joinGame	77,374
Battleship	depositBet	56,878
Battleship	commitBoard	55,330
Battleship	shoot	103,623
Battleship	confirmAndShoot	119,912
Battleship	boardCheck	253,046 - 1,509,487
Battleship	withdraw	35,513
Battleship	report	78,942
Battleship	verifyReport	85,627
Battleship	forfeit	75,935

Table 1: Gas cost of functions

A few notes on these results:

- The method **createGame** is very expensive. This is expected, since it needs to deploy a new Battleship contract on the blockchain, an operation that requires a lot of gas.
- Since **boardCheck** only verifies the cells that haven't been shot by the opponent, its cost depends greatly on how long the game was. If the game is won in just 10 shots the cost will be the greatest, if the game is won after missing all the shots the cost will be the smallest.

The test also calculates the total cost of the smart contracts for a game where each player misses all the guesses. Adding together the gas used by each function used, considering that some functions are called more than once (e.g. `confirmAndShoot` is called 127 times) and that the cost of `boardCheck` greatly depends on the shots taken, the total gas used comes out to be: **16,017,323**.

## 4 User manual

### 4.1 Running instructions

To run the project:

1. start a local blockchain instance with Ganache UI on port 7545
2. from inside the `ethereum-battleship/` folder, run `npm ci` and then `truffle migrate`
3. from inside the `ethereum-battleship/frontend/` folder, run `npm ci` and then `npm run dev`

This will automatically open the homepage of the game in a browser. Open the same page in a different browser to play locally with two or more players. In both browsers you'll need to setup a different MetaMask account on the correct network.

### 4.2 Tests

To evaluate the gas cost, run `truffle test` from inside the `ethereum-battleship/` folder. The outputs will be printed in the console.

### 4.3 Demo

If everything was setup correctly, you should be greeted by the following homepage. From here, after connecting to MetaMask, the player can choose to create a new game or join an existing one, either by using a game ID or randomly.

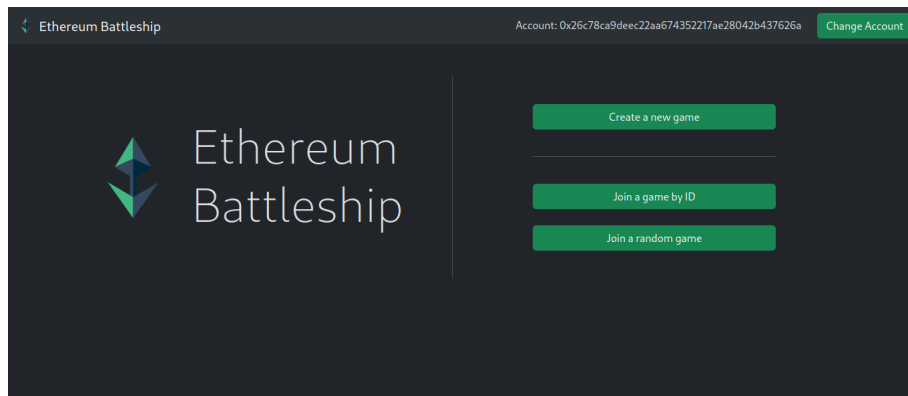


Figure 1: Homepage

We start by creating a new game. From this page, the creator will choose the bet for the game.

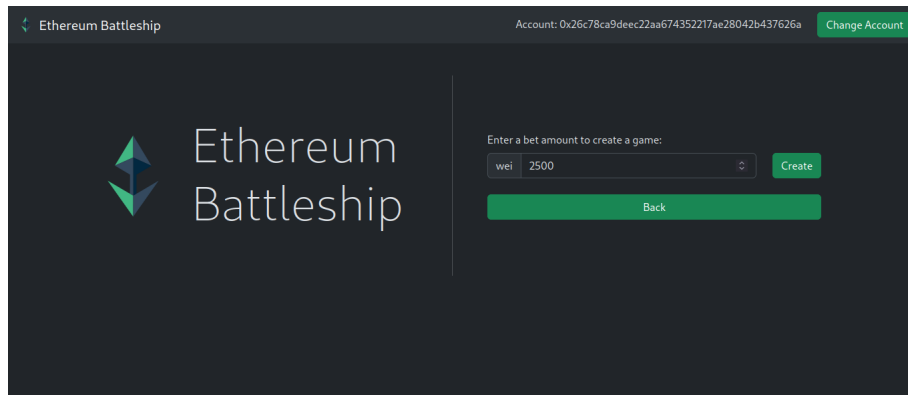
The screenshot shows the 'Ethereum Battleship' web application. At the top, there's a header with the app name and a user account address: 'Account: 0x26c78ca9deec22aa674352217ae28042b437626a'. A 'Change Account' button is on the right. The main area is split into two sections. The left section features the 'Ethereum Battleship' logo. The right section is titled 'Enter a bet amount to create a game:'. It contains a text input field with 'wei' as a unit and '2500' as the value. To the right of the input is a green 'Create' button. Below the input field is a green 'Back' button.

Figure 2: Creating a game

After creating the game, the creator will be redirected to a waiting page, where the game ID can be copied to be shared. Staying in this page will also redirect the creator to the deposit page when an opponent joins the game.

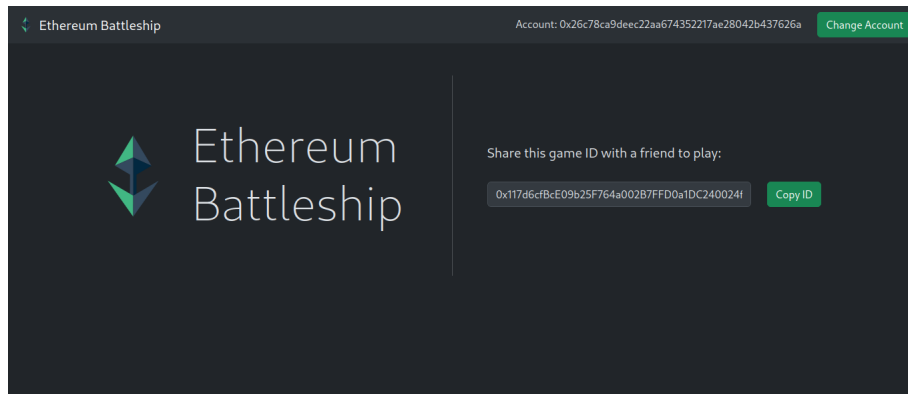
The screenshot shows the 'Ethereum Battleship' web application in a waiting state. The header is identical to Figure 2, showing the app name and the user account address. The main area is split into two sections. The left section features the 'Ethereum Battleship' logo. The right section is titled 'Share this game ID with a friend to play:'. It contains a text input field displaying a long hexadecimal game ID: '0x117d6cf8cE09b25F764a002B7FFD0a1DC240024f'. To the right of the input field is a green 'Copy ID' button.

Figure 3: Waiting for an opponent



Meanwhile, the second player can find a game using a shared game ID or randomly. After having found a game, they will be redirected to the following page, where they can check the game's information (the ID, creator and bet) and decide to join it or not.

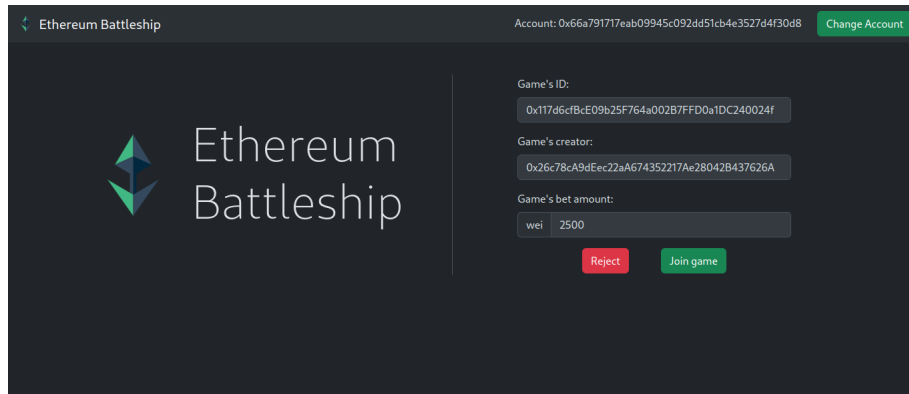


Figure 4: Joining a game

When the second player joins the game, both players will be redirected to the deposit page. Here both players have to deposit the game's agreed bet before proceeding.

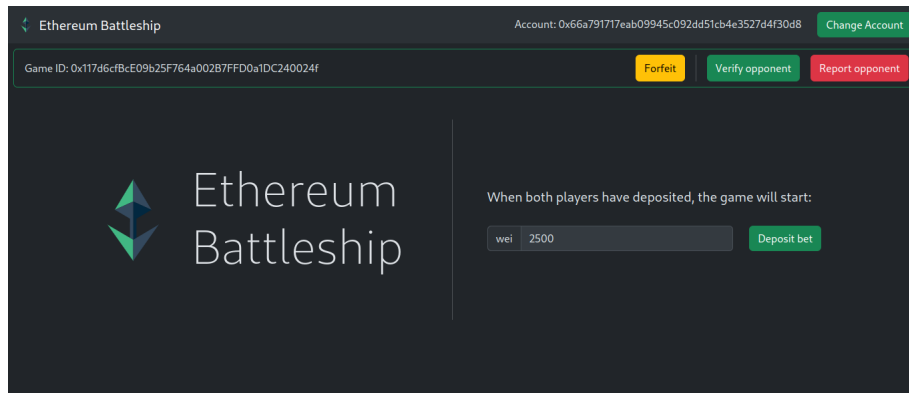


Figure 5: Depositing the bet

After that is done, the game will begin, starting with the placing of the ships. Both players will need to place the required number of ships and, when they are done, they both have to commit their board.

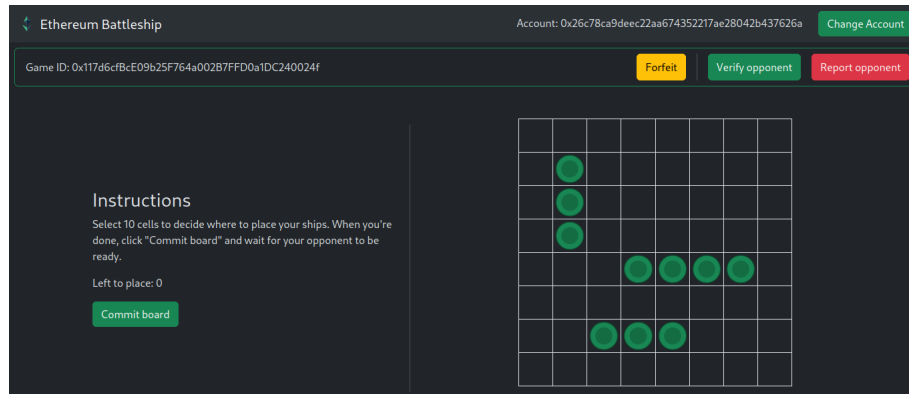


Figure 6: Placing ships

They will now enter the attacking stage, taking turns shooting at the opponent's board by clicking on the the desired cell. The color of the cell will depend on a few factors: green for the position of your ships, red if a shot sunk a ship, blue if it missed or yellow if it still needs to be confirmed.

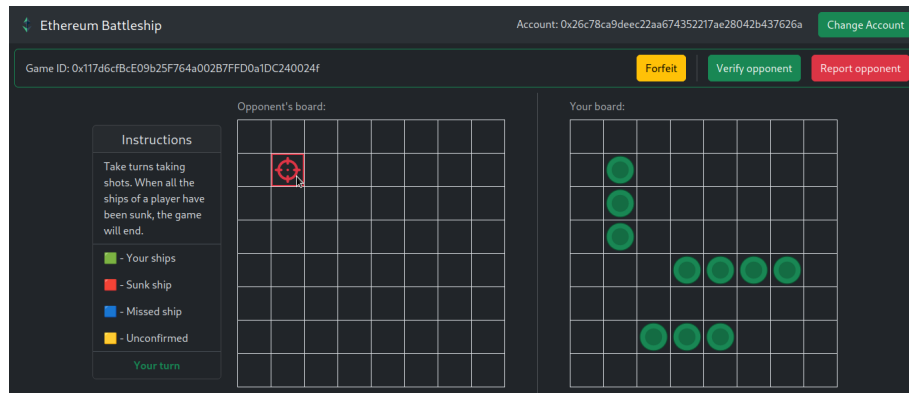


Figure 7: Playing the game - Beginning



Figure 8: Playing the game - Shots taken

When one of the two players manages to sink all the opponent's ships, they will be declared the winner. To be able to withdraw the prize, the winner still needs to send its board for verification.

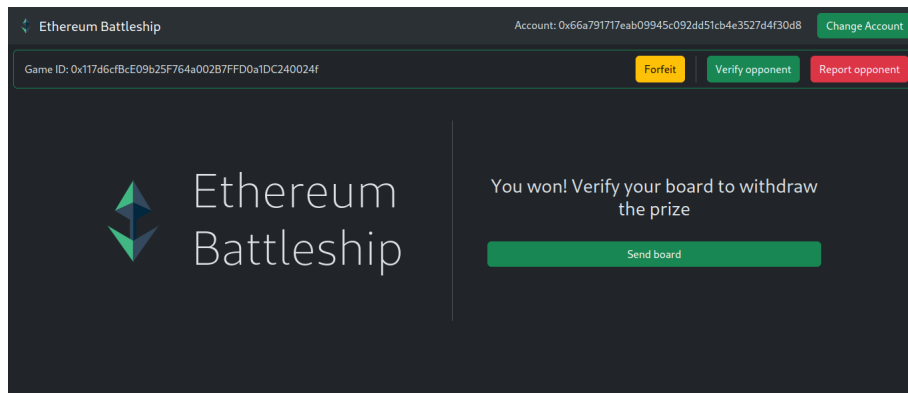


Figure 9: Winner verification

If the board is verified, the winner can withdraw the prize. If something goes wrong in the verification, it means that the supposed winner was cheating, so the opponent will be declared the actual winner.

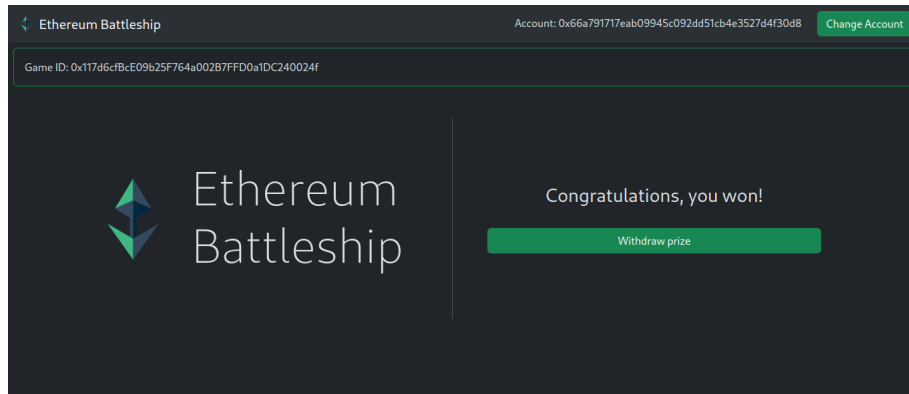


Figure 10: Withdrawing the prize