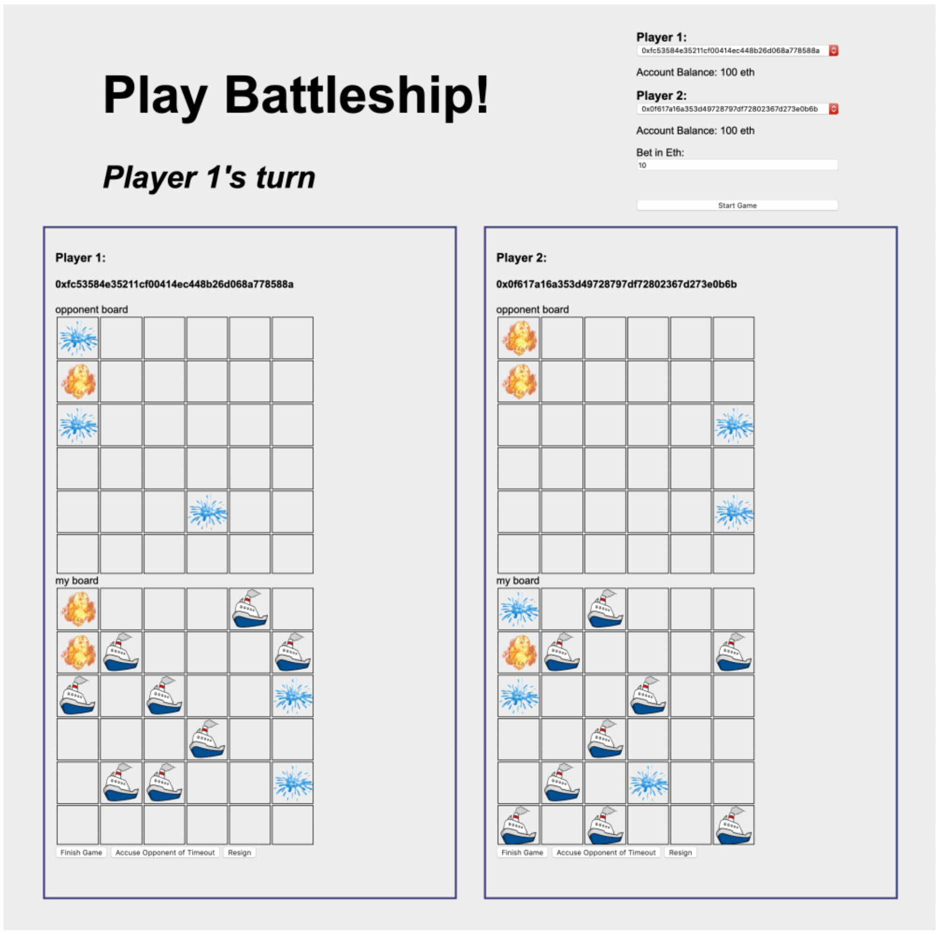
CS251: Cryptocurrencies and Blockchain Technologies **Fall 2020**

Programming Project #3

Due: 11:59pm on **Wed., Nov. 4, 2020**
Submit via Gradescope (each answer on a separate page) code: **92ZNG8**

For this project, you will be working on a web client and a Solidity contract that will enable two players to play a state channel battleship game. A player interacts with the Solidity contract once at initialization time to start the game, once (if they won) at the end of the game to claim a win, and by choice when necessary to claim the opponent is cheating or to accuse the opponent of taking too long to make the next move.

Throughout this project, you are expected to look out for possible ways a malicious user could cheat this system and write code that defends against these attacks. You do not have to defend against all possible attacks, but you should consider attacks that we do not require you to defend against. You will also answer some written questions about security considerations that account for attacks we have not protected against in the code. We will guide you with questions to think about writing good defensive code. Let's get started!



Player 1: 0xfc53584e35211cf00414ec448b26d068a778588a
Account Balance: 100 eth
Player 2: 0x0f617a16a353d49728797df72802367d273e0b6b
Account Balance: 100 eth
Bet in Eth: 10

Player 1:
0xfc53584e35211cf00414ec448b26d068a778588a

Player 2:
0x0f617a16a353d49728797df72802367d273e0b6b

1 Overview of the Battleship Game

Battleship (the board game) is a guessing game played by two players who cannot see each others' boards until the end of the game. Both players normally place 5 ships of varying lengths on their 10x10 board, which simulates the ocean. Then they take turns making guesses about the locations of their opponents' ships. To guess, a player tells the opponent the [i,j] coordinates they wish to check. If any of the opponent's ships are at that location, the opponent replies that they have been hit. Otherwise the opponent replies that they have missed.

In the board game setting, players can cheat by saying they have placed 5 ships when they have actually placed none. They can also cheat by lying about hits and misses when the opponent makes a guess.

In our implementation of Battleship, the board is 4x4 (a total of 16 squares), and each player must place exactly 2 ships. Each ship takes up exactly one square. To prevent cheating about the number of ships, players interact with a Solidity contract to commit to their initial board states. During game-play, they interact off-chain, without touching the contract. They can verify hits and misses locally. If a player suspects the opponent of lying about a hit or miss, OR if the player suspects the opponent of having left (taking too long to respond), they can use the Solidity contract to make and resolve these accusations.

2 Getting Started

1. If you haven't done so for the previous project, you'll need to download and install Node.js from `https://nodejs.org/en/`. Choose the LTS version (the one on the left).

2. If you haven't done so for the previous project, run `npm install -g ganache-cli` to install the Ganache CLI, which we will use to simulate a real Ethereum node on our local machines. Then, run `ganache-cli` to run the node. You can stop the node at anytime with Ctrl-C.

3. Download the starter code from the class website and open it in your favorite IDE or text editor (something like Sublime Text, Atom, or Visual Studio Code works nicely). Familiarize yourself with the code base. For this project, you are expected to fill in `contract_starter.sol` and `BattleshipPlayer.js` files. You may add helper functions to these two files if needed. There are also helper functions in `battleship_game.js` and `utils.js` that are used by the UI. You can read them for familiarity.

4. Open `https://remix.ethereum.org` in your web browser. In the 'Deploy & Run Transactions' tab, set the environment to 'Web3 Provider', click 'Ok' when prompted, and then set the 'Web3 Provider Endpoint' to `http://localhost:8545` - this should be the default. This is where you will develop your smart contract (which you will write in Solidity). In the 'Solidity Compiler' tab, set the compiler to version 0.6.6+[latest available commit].

5. Open `index.html` in your browser (works best in Chrome) to access the client interface where you can test your game implementation. On this page, you can pick 2 players by address and specify how much each of the players are betting ("`ante`") to play this game, specified in ETH (note that the account balance for a particular user must be larger than or equal to the bet they are making for it to be allowed to play the game). There are two views of the game: the view on the left consists of the information that Player 1 is given. Note that Player

1 doesn't know the placements of the ships of the second player, and vice versa for Player 2, but you can see both boards side by side. This should make testing your implementation easier.

6. It's very helpful to also open your browser's JavaScript console, so that you can see `console.log()` messages and error messages. If everything so far is working, you should see no errors in the console (you can safely ignore warnings).

7. Implement code for the requirements outlined below. When you have your contract, compile and deploy it, and then **update the contract hash and ABI** in `BattleshipPlayer.js`. The ABI can be copied to the clipboard from the 'Solidity Compiler' tab, and the contract hash can be copied from the 'Deploy and Run Transactions' tab. Note that the contract hash is *not* the transaction hash of the transaction that created the contract. If you find that some behavior works once and errors on subsequent tries, re-deploy your contract and copy the new hash. This is an indication that state has not been cleared in solidity (one of the todos).

3 Lifetime of Your State Channel Battleship Game

Before you begin, read all parts carefully. Some parts require you to implement multiple functions, but the total amount of code will be small. The **clear_state** and **is_game_over** functions mentioned in 3.5 may be useful for all the parts.

3.1 Starting a Game

At the beginning of the game, players place their bets by interacting with the contract. Both players bid the same amount by the nature of our UI. After bidding, they place their battleships and generate a commitment to their initial board state. To implement this, you will have to modify the following in `BattleshipGame.js`:

- **constructor**: Store any state you would like to keep locally, and initialize event handlers (you can do this in part 3.4).
- **place_bet**: Interact with the contract to place bets.
- **initialize_board**: Most of this is done for you. Add interaction with the contract to store initial board states. See section 3.2 for what board state means.

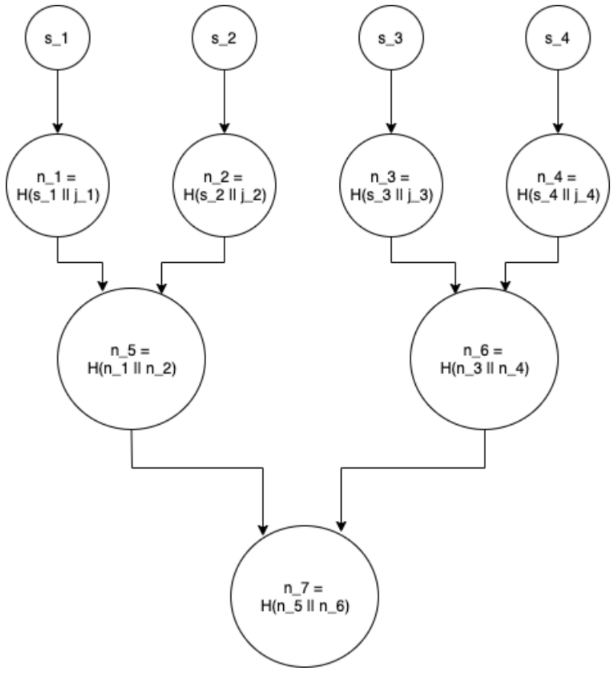
You will also have to modify the following in `starter_contract.sol`:

- **store_bid**: The first player to talk to the contract sets the required bid to play the game. The second player must bid that amount, or the game will not start. Refund excess money back to the second player if they bid too much.
- **store_board_commitment**: Store any state necessary to verify later that the players did not lie about their boards.

3.2 Off-Chain Game Play

You do not need to write any code for this part. After the game is initialized, players make guesses by clicking on the version of the opponent's board that they can see on their turn. Hits are seen as explosion images and misses are seen as splash images. All of this interaction is off-chain.

Since board states are kept secret in a real game, we use a Merkle tree commitment scheme to report the state of the board to the contract so that no other entity can tell where your ships are. Consider the following scheme:



Assume Player A is playing with an N x N board, with N^2 cells in total, and placed some k number of ships on it. For each cell of the board, we create a leaf node n_i for $i \in [1, N^2]$ using $n_1 = H(s_1||j_1)$, where s_1 is the state of the first cell, and j_1 is a randomly chosen nonce. The state of a cell is either a 1 (denoting presence of a ship) or 0 (denoting an empty cell). Based on these leaf nodes, we construct a Merkle Tree. For example for $N = 2$ (4 cells on the board in total), a leaf node would be: $n_1 = H(s_1||j_1)$, and a node at level 1 would be: $n_5 = H(n_1||n_2)$. The diagram above shows what the Merkle tree for this example would look like.

To prove that a ship is at a particular location in the Merkle tree, the owner of the ship must provide a proof of inclusion. This proof consists of the hashes of nodes along the path from the root to the leaf, along with the hashes of sibling nodes at each level.

Consider what happens if we omitted the nonce values from the hash calculations for the leaf nodes. Given the Merkle root, it would be trivial to brute force the board state since there are only N^2 tiles (for some small value of N) and the tiles have binary states.

3.3 Accuse Opponent of Cheating

At any point, either player may accuse the other player of cheating. We will define **cheating** as lying about the location of a ship by providing a faulty proof. The client-side implementation of a

Merkle tree already checks that each proof in response to a player's guess is verified. Your task in this section is to implement the same check using the contract. To implement this, you will have to modify the following in `BattleshipGame.js`:

1. **accuse_cheating**: Send the contract what it needs to verify if a player is cheating.

You will also have to modify the following in `starter_contract.sol`:

1. **accuse_cheating**: Use what the client sends to verify that the opponent is not cheating. If the opponent is cheating, send the total winnings to the accuser.

3.4 Accuse Opponent of Leaving

At any point, a player may accuse the other player of having left - if the opponent is taking too long to make a move, for example, the player may accuse them of leaving. This accusation should trigger an event in Solidity that must be addressed by the opponent within a time limit of **1 minute**. If the time runs out, the accuser may make a second call to the contract to claim the total winnings.

To implement this, you will have to modify the following in `BattleshipGame.js`:

1. **accuse_timeout**: Notify the contract that you would like to accuse the opponent of leaving.
2. **handle_timeout_accusation**: If you have been accused of leaving, interact with the contract to indicate you are still playing. This will keep the winnings from being distributed when the clock runs out.
3. **claim_timeout_winnings**: If you accused the opponent and they did not respond in time, interact with the contract to claim the winnings.

You will also have to modify the following in `starter_contract.sol`:

1. **claim_opponent_left**: Use Solidity events to start a timer. See [here] for an introduction (also linked under References). The latest version of web3 has some minor differences from this syntax.
2. **handle_timeout**: If you have been accused, do what is necessary to keep the timer from running out.
3. **claim_timeout_winnings**: If the timer has run out for the opponent, claim the winnings.

3.5 Forfeit

At any point, either player may forfeit. Having locked their bids in the contract at the start of the game, a player who forfeits loses their money immediately, and the opponent wins. This happens regardless of cheating, lying about ship counts, etc. To implement this, you will have to modify the following in `BattleshipGame.js`:

1. **forfeit_game**: Tell the contract you're forfeiting.

You will also have to modify the following in `starter_contract.sol`:

1. **forfeit**: Send all winnings to the opponent and end the game.

3.6 Claim You Won

Only at the end of the game do the players open the commitments necessary to show that they have won the game - the placements of the two ships on their own board AND the two guesses (with proofs) that resulted in hits on the opponent's board. If the winner cannot provide these proofs, they cannot win the game. To implement this, you will have to modify the following in `BattleshipGame.js`:

1. **claim_win**:

You will also have to modify the following in `starter_contract.sol`:

1. **check_one_ship**: Check that one ship is where it is supposed to be, given a merkle proof. Use this to indicate your ship placements and winning guesses.
2. **claim_win**: If you have checked the necessary ships, this function should send you all the winnings.
3. **clear_state**: Clear state and set the game to be over.
4. **is_game_over**: Getter function so you can check in web3 if the game is over.

4 Written Questions

Please answer these written questions in a pdf file to be submitted separately. Limit your answers to 3 sentences maximum:

1. Suppose player 1 places less than 2 ships on the board but never lies about hits or misses. Can player 2 get their money back? Why/why not?
2. Why do we not restrict players from placing more than 2 ships on their board?
3. We do not have a mechanism to accuse the player of placing less than 2 ships on the board. How might you implement this?
4. Can you think of attack scenarios or specific vulnerabilities in any of the given code that you could not defend against?

Optional Feedback. How can we make this project better in the future?

5 Submission

Submit `BattleshipGame.js` and `starter_contract.sol` to the Gradescope assignment for Project 3 Code.

Submit your pdf of written question solutions to the Gradescope Assignment for Project 3 Written.

You are allowed to work with 1 project partner (team size ¡= 2). Don't forget to add your project partner to your Gradescope submissions. If you are using late days for this project, make sure that both partners have enough late days remaining according to class policy on the website.

6 References

- Wikipedia page for the battleship game: https://en.wikipedia.org/wiki/Battleship_(game)
- State Channels: https://arxiv.org/pdf/1702.05812.pdf
- Solidity Events - https://programtheblockchain.com/posts/2018/01/24/logging-and-watching-solidity-events/