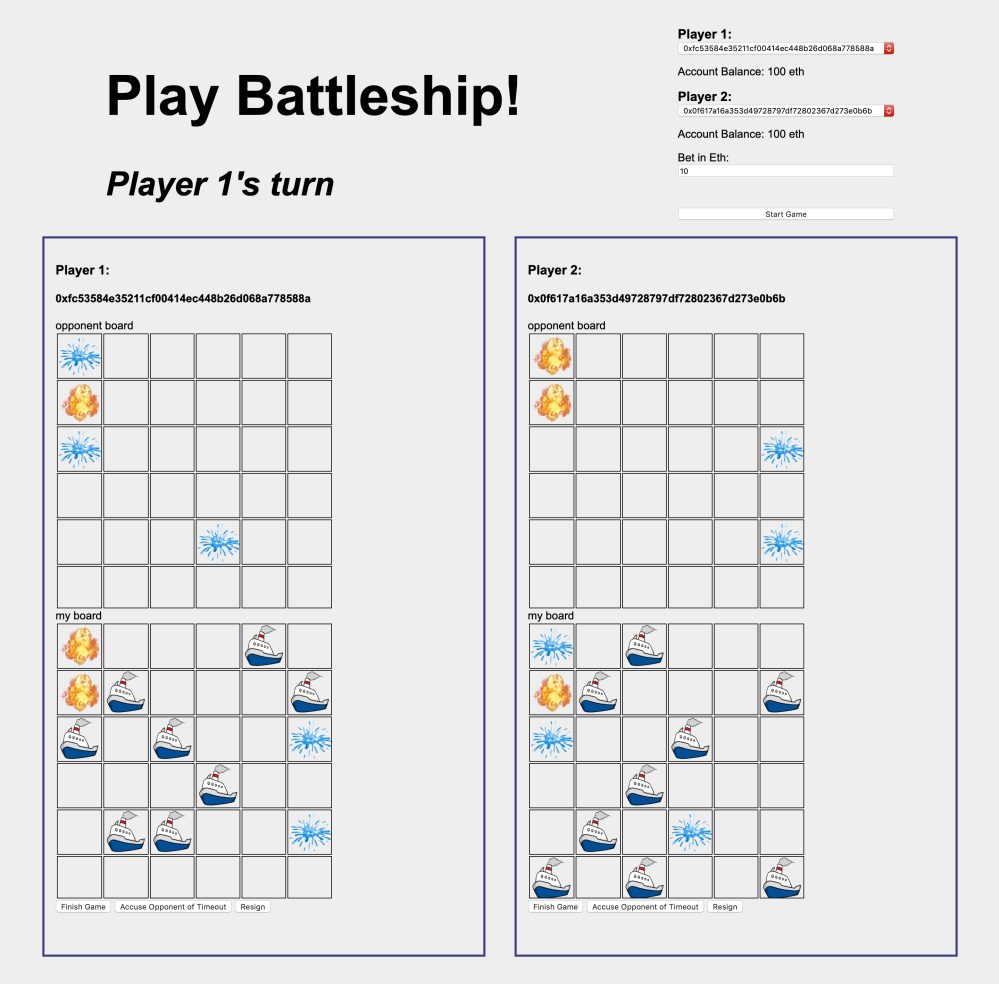
CS251: Cryptocurrencies and Blockchain Technologies **Fall 2019**

Programming Project #3

Due: 11:59pm on **Mon., Nov. 11, 2019**
Submit via Gradescope (each answer on a separate page) code: **MG7EP3**

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, possibly once at the end of the game, and possibly once if the other player cheats or disappears. Otherwise, the players exchange messages amongst themselves, and do not interact with the contract. Specifically, if the players are honest, there are only three transactions sent to the contract.

Throughout this project, you are expected to look out for possible ways a malicious user could cheat this system and implement defenses against these attacks. We will guide you with questions to think about some of the attacks that we came up with. Let's get started!



1 Overview of the Battleship Game

If you have never played Battleship before, we suggest that you familiarize yourself with the setup of the game first by reading up on it. Here is a quick summary. Battleship is a strategy type guessing game for two players. Two players set up their boards by placing some number of battleships in a grid. The board state is kept secret from the other player. Players alternate turns calling "shots" at the other player's ships, by guessing some x and y coordinate. The objective of the game is to destroy the opposing player's fleet.

In the original game, the players need to place a certain number of ships of different sizes. Note that for this project, you don't have to worry about controlling the number of the lenghts of the fleets. You can count ships as single cells in our game, and you only dictate that each player places ships to occupy some set number of squares on the grid (the occupied squares don't have to be consecutive for example).

The players exchange moves using state channels amongst one another, off-chain, without calling into the contract. The advantage of a state channel implementation such as this is that it significantly saves gas costs to let the game only interact with the contract at the initialization and the conclusion stages of the game. See the references listed at the bottom of this spec for a refresher on state channels.

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 `mycontract.sol` and `BattleshipPlayer.js` files. There are places marked with functions to modify - please do not modify any of the other code. Feel free to add helper functions.

4. Open `https://remix.ethereum.org` in your web browser. In the 'Run' 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).

5. Open `index.html` to access the client interface where you can test your game implementation. On this page, you will see a list of addresses that you can choose from for each of the two players, a text field where you need to 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 fleet placement of the second player,

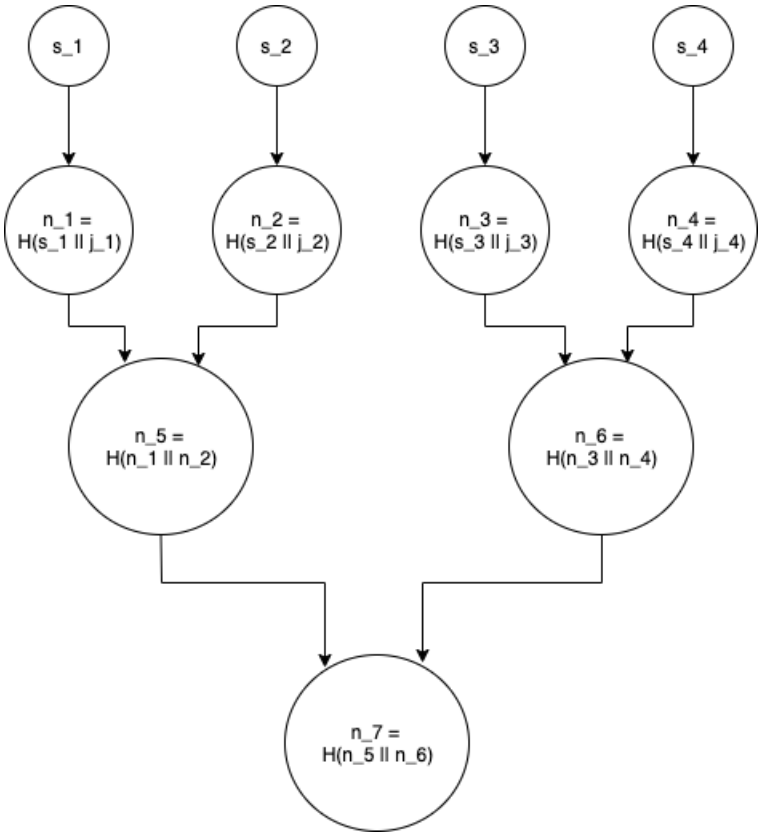
and same for Player 2. For the purposes of this project, we allow you to view these boards side by side so that you can test your implementation easier.

6. It's very helpful to also open your browser's JavaScript console, so that you can see error messages (there's a link on how to do this at the end of this document). If everything so far is working, you should see no errors in the console (you will probably see a warning; this is fine, see the the notes at the end for more details).

7. Implement code for the requirements outlined below. When you have your contract, deploy it, and then **update the contract hash and ABI** in `BattleshipPlayer.js`. The ABI can be copied to the clipboard from the 'Compile' tab, and the contract hash can be copied from the 'Deployed Contracts' panel in the 'Run' tab. Note that the contract hash is *not* the transaction hash of the transaction that created the contract.

3 Lifetime of Your State Channel Battleship Game

At the beginning of the game, both players should make calls to the contract from the client to initialize a game using the functions `constructor` and `initialize_board` in `BattleshipPlayer.js`. This communication should include the two players' addresses and the ante (i.e. bid) amount that they both agreed to. The players should send one another hiding commitments of their initial board states. Since board states are kept secret in real life battleship, we ask that you implement a commitment scheme to report the state of the board to the contract so that no other entity can tell where your fleets 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 fleets 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 this Merkle tree would look like for this example.

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.

For our simplified battleship game project, you will be playing on a 6x6 board (N=6) and you are required to place 10 battleships (k=10). Make sure that you enforce in the smart contract to place 10 secret battleships (as mentioned at the beginning, you don't need to worry about controlling the number of the lengths of the fleets. In this project, count ships as single cells on the gameboard). **Hint:** Notice that at the beginning of the game the board states are kept secret. Only at the end of the game do the players open their commitment to the initial board, which is when you can enforce that each player placed k ships on its board. If not, the player forfeits the game.

Once the game initialization step is finished, the game starts! When Player A sends a move (x, y) (intending to shoot at coordinates x, y) along with any other metadata necessary to Player B, B verifies A's information, opens the commitment for that cell, and sends a Merkle proof to A indicating whether A has shot one of B's battleships or not, along with any additional data if needed. You should think thoroughly what information would be needed for Player A and B to exchange in this communication between each move, with the goal to securely verify the move and inform the result, as well as to prevent any malicious attacks outlined in the Implementation Requirements section.

The game also has an additional punishment rule: if and when one player cheats or disappears, any one of the two players can submit a transcript of the moves (along with some metadata of your design) to the contract. The contract should enter a waiting stage. At this point, the opposing player has a chance to return to the game, to submit their move. At the end of the waiting period, the contract pays the total amount of bids to the winning player. Alternatively, if one of the players wins the game, they can make a call to the contract to verify this victory and claim the reward. The contract should incentivize an honest and correct conclusion to the game, which can be achieved by returning a small amount of the original ante to the losing player at the end of the game. This will incentivize the losing party to sign the final message conceding the game.

4 Implementation Requirements

Your submission should be able to interact with the Solidity contract to initialize a new game amongst two players with a specific bid amount. The contract should withdraw this bid amount from both players' account balances before the game starts. Then, the two players should exchange moves and some metadata about each of these moves. At the end of the game, the winner or the loser should be able to interact with the Solidity contract to end the game. At the end of the game, the Solidity contract should send the total sum of the bets to the winner. If at any point, one of the players attempts to cheat or disappears, then the other person is declared the winner.

Throughout this project, you should think about how you are defending against malicious users.

Here are some cases to consider. Note that the list below is not a comprehensive list of issues that your code will have to handle gracefully. For each of the attacks you are considering, we ask that you discuss them in the design doc you are submitting. See the Submission section for more details.

4.1 Stage 1: Game Initialization

1. Consider what happens if Player A decides to start a game against Player B and makes a call to the Solidity contract to pay the bid amount. What happens if Player B never makes a symmetric call to the contract to pay her amount? How do you ensure that Player A gets her money back? What should the contract do in this case?

2. What happens if Player A submits a game against Player B with a 10ETH bid, but player B's corresponding bid is only 1 ETH? You can set a game that takes the minimum of the two bets from each person, and refund any differences back to the respective player. Thus in our example above, the game will involve taking 1ETH stake from each person, and refunding 9ETH back to A at the start. You are free to specify any other scheme. Also be aware of no negative amount of antes, as that would be stealing money from the pot!

3. How should the contract keep track of the ongoing games? For this project, you can assume that each player will play only 1 game at a time, so you don't have to handle multiple games for a single player.

4. How can you prevent cheating in the set up stage? Consider the case that while setting up the game, Player A places more than 10 battleships on his board. How can you prevent and penalize this type of a behavior? **Hint:** think about what should happen at the end of the game when A wins and declares this victory to the contract. What checks should the contract have in place to make sure that the players didn't cheat during the set up stage?

4.2 Stage 2: Exchanging Moves

1. Once players have initialized their boards at the beginning of the game, how can you keep them accountable that they won't modify the placements of their fleets during the game? Consider the case where A places a correct number of fleets at the beginning of the game but removes them later on. **Hint:** think about how the players should declare the state of their boards to the other player and to make it impossible to change it later on. At the same time, the board states should be kept secret unless the players have "opened" certain squares by taking shots at those coordinates during their turn or until the very end of the game. Think about how you might be able to leverage Merkle commitments and proofs for this implementation.

2. How can you prevent a user from "forking", i.e. claiming that they submitted a different move in the past or insert past moves into the game history? I.e. how can you make sure that the exchanged moves are ordered and the players cannot cheat? You might want to think about what metadata the users should pass over along with their (x,y) coordinates, such as creating "sequence number" (counter) for each move in turn.

3. What if the opponent is not making a move? Think about introducing a timeout feature. For example, you can post to the blockchain if you believe that your opponent will time out.

You send the blockchain his most recent move and your most recent move. The contract then enters a special state where it waits for the opponent to post their next move to the blockchain before he is timed out. If the opponent is timed out, then the contract declares you the winner and after receiving the relevant data from both parties, the contract should send you the money. Else if the move is submitted in time, then the game continues back in the state channel.

4. How can Player A verify that the other move indeed came from Player B? Similarly, at the end of the game, how can the solidity contract verify that the series of moves submitted were indeed created by Players A and B (i.e. Player A didn't manufacture a fake game transcript)? You might want to think about how to leverage signatures for this requirement. How can you create signatures and verify them? Who should be doing these checks?

4.3 Stage 3: Game Conclusion

1. Once the game is over (either because a winner has been declared, or someone has timeout or any other ways of cheating you can think of), the game is then ended and will be posted back to the contract.

2. Design yourself what information is required at this stage for the contract to verify the result, and make the eventual transfer. We have included a few helper functions that will aid you in the process. Note the final state of the game needs to be verified, and the loser needs to sign a resignation to declare losing. The players must be incentivized to conclude the game correctly. How can you incentivize the losing party to sign the final message conceding the game? As otherwise we will always need to go through a timeout process. Feel free to design the final contract's allocation of the ante.

5 Refresher on Crypto Primitives

5.1 Digital Signature Schemes

Digital signatures can help verify the identity of the sender of a message. Recall that a digital signature scheme consists of three function: (G, S, V). G is the generator method that outputs a public key and private key. S is the signing method that outputs a signature given a secret key and message pair (valid message in message space M). V is the verification method that outputs true or false given a public key, message and signature tuple.

$$G() \rightarrow (Pk, Sk)$$
$$S(Sk, m \in M) \rightarrow \sigma$$
$$V(Pk, m \in M, \sigma) \rightarrow true/false$$

For this project, we provided you with signature methods that you can use in your client and contract codes. You can find these methods in the `utils.js` file.

6 Practical Development & Debugging

- On your client code, you can use `console.log("debugging info here")` to debug your implementation. The strings that you print using this command will appear on the console.

You can access by right clicking on your browser and choosing "Inspect" > "Console".

- We will be posting an up-to-date listing of all clarifications and advice on Piazza. Please follow that post to get the latest information as we work any issues with the assignment.

7 Submission

We will be using Gradescope for submission. Please submit a zipped version of your `mycontract.sol` and `BattleshipPlayer.js`, along with your `DESIGN_DOC.txt`.

Your design doc doesn't need to be longer than 3 pages. This is a way for you to explain your implementation to us to ensure you get the points you deserve. In your design doc, you should aim to answer the following questions:

1. What scheme did you come up with for the communication between two players during the game? What pieces of information do users send to one another and explain why each are necessary to prevent malicious behaviors.

2. For each of the bullet points from Implementation Requirements section that you addressed in your code, what was your strategy in preventing this problem?

3. Are there other defenses that you implemented in your code against malicious behaviors that weren't mentioned in the spec? Were there any attack scenarios that you couldn't defend against?

You are allowed to work with another project partner. Don't forget to add your project partner to your Gradescope submission. If you are using late days for this project, make sure that both partners have enough late days remaining.

8 References

- Wikipedia page for the battleship game: https://en.wikipedia.org/wiki/Battleship_(game)
- State Channels: https://arxiv.org/pdf/1702.05812.pdf