Introduction to Cryptocurrencies - Ex3

Due: Tuesday, January 6th 23:59

Goals This exercise is intended to make you learn solidity. In part I, you'll deploy a wallet contract and attack it using a re-entrancy attack. In part II, you'll implement a rock-paper-scissors game.

Setup
You will need:

- Remix IDE at `https://remix.ethereum.org`
- Hardhat.
  - On your own machine, you can install hardhat using npm (select default options):

    ```
    npm init
    npm install --save-dev hardhat
    ```

    initialize a new harhat project (Create an empty hardhat.config.js):

    ```
    npx hardhat init
    ```

    and run the node with:

    ```
    npx hardhat node
    ```

Good places to see solidity syntax and understand the language:

- solidity-by-example (this address is blocked from JCE networks, so you may need to use a different internet connection to access it)
- the solidity documentation

See additional links and screenshots in the slides from class.

Part 0 (warm-up)

1. Deploy the contract `Wallet` from the file `VulnerableWallet.sol` using Remix. Do this in a local Hardhat network that you connected Remix to.
2. Deposit some money to the wallet. Do this from several different accounts that Hardhat provides.
3. Check that you can transfer money from the wallet back to one of the addresses.

There is no need to submit anything for this part.

Part I: Reentrancy

The smart contract in `VulnerableWallet.sol` has a reentrancy bug and can be attacked. Your job is to perform such an attack.

- Inside `WalletAttack.sol`, implement an attacking contract that will exploit this vulnerability. The exploit should run after the `WalletAttack` contract is deployed, and its exploit function is called with 1 ether of value.
- To be successful, you must withdraw at least 3 ether from the victim contract. This means you'll need to test this out with a victim wallet that has at least 3 eth deposited to other accounts so that there is sufficient money to steal. This is in addition to the 1 ether that the exploit function receives when you call it.
- There is no need to change the amounts stolen based on the amounts inside the wallet. Just be sure to steal at least 3 ether. Send all funds to the attacker's address (whoever called the exploit function)

Misc. Hints

- To call one contract (like the `VulnerableWallet` contract) from another contract (the attacking contract) it is best to use an interface. The Wallet interface has been defined for you inside `WalletAttack.sol` (it is called `WalletI`). You can then cast the address to an IWallet, and call some function there.

  ```
  Wallet(wallet_address).some_function();
  ```

- [**This part is optional:**] You can use Remix to write and test your code, or alternatively, use a script in python for compilation, deployment and execution. You can see an example in the file `deploy.py` within the `deploy_example` directory.
  This will require you to install (via pip):

  - **py-solc-x**: A python module that wraps the solidity compiler
  - **web3**: A python module used to interact with Ethereum nodes (see related documentation)

  We will use these tools more extensively in future exercises, so it is helpful to get familiar with them.

  In this case it is advisable use VSCode (instead of Remix) as your IDE with the solidity extension (it is open-source & free to download and use). Cursor or Antigravity are also good fit.

Part II: Rock-Paper-Scissors

In this part, you'll implement the Rock-Paper-Scissors game as a solidity contract. The contract will support multiple games.

The contract's API is given in the file `RPS.sol`. Be sure to implement it fully. Be sure not to change the API (feel free to add state variables, functions, etc., including public ones, but do not change the signatures of existing external functions, or the constructor).

The main challenge in the design of the game is to ensure that players cannot cheat. Since the moves in rock-paper-scissors needs to take place simultaneously, but blockchains only allow one transaction at a time, we'll use commitments to allow players to enter moves in a way that is hidden from their counterparts. (See Bit commitment in the random oracle model here). The players will then reveal their commitment and the winner will be determined.

Depositing and Withdrawing Money

The RPS contract can receive money from any source via a direct transfer (without an explicit function call), which is later used for betting on the outcome of games.

Users with money in the RPS contract can withdraw it, as long as these funds are not locked in an ongoing rock-paper-scissors game.

Phases of the Game

After players deposited money to the contract, each game proceeds according to the following phases:

- Each player commits to a move using the `make_move()` function. This also includes locking some of the balance of this player as a bet (the first player to move determines the bet size, and the second player needs to have enough funds to match it).
- Once two moves have been entered, each player then reveals his move via the `reveal_move()` function.
- Once both moves are revealed, the game is over, and the winner gets all the rewards. In case of a tie, the bets are returned to the players.

The process above describes a "good" flow of the game, in case both players are honest. However, one of the players may try to cheat, or stop responding. In this case, the other player should always have the means to get the bet back and to end the game:

- If one player committed a move, and the other player failed to respond, then the first player can cancel the game. The game cannot be cancelled later. (Food for thought: what's the problem with canceling after the other player committed his move, but before he revealed it?)
- If one of the players revealed his move, and the other player does not reveal within a predefined reveal-period, then the revealing player can decide to cancel the game. In this case he wins all of the money, including the money wagered by the non-revealing player. (Food for thought: Why is this the case? Can't we just cancel the game and give each one their funds back?)

Each of the calls above identifies the game that is being played using a `gameID` – an arbitrary number that the players chose. It is additionally possible to see the state of the game using the `getGameState()` function.

Committing to a Move

In order to implement a simple commitment scheme, we will simply allow the players to send the hash of the move with some long random key as a commitment. The commitment is later checked by getting the move and the key and checking that the hash matches. The RPS contract includes a function that allows you to compute such hashes to check commitments, using solidity's Keccack256. You are also given accompanying python code (`commit.py`) that will let you create commitments of this sort in a way that matches the internal solidity representation. To run this code you'll need to install the packages `web3` and `hexbytes` using pip.

Submission

Submit your files in a single zip file called ex3.zip. The structure of the zip file should be flat (with no internal directories), it should contain only the files WalletAttack.sol and RPS.sol. It should additionally contain a README file (not README.txt). The README file should include your name and ID as well as a brief description of the exercise. Do not submit files that we provided such as VulnerableWallet.sol.

Your code will be auto-tested. You are naturally allowed to modify the files we provided as long as you do not break the API that we describe (i.e., you can add more methods, structs, enums, and instance variables to the contracts in each of the files above.

Good luck!