Project 2: “Smart” Contracts

This project is due on **Tuesday, February 26** at **6 p.m.** and counts for 10% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. Late work will not be accepted after 24 hours past the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; you will work in **teams of two or three** and submit one project per team. Please find a partner as soon as possible. If have trouble forming a team, post to Piazza’s partner search forum.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else’s solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via **Dropbox**, following the submission checklist below.

Introduction

In this project, you will write an Ethereum smart contract to play a simple game of Rock/Paper/Scissors, and deploy it on the Ropsten Test Network. You’ll also exploit an existing smart contract to steal (testnet) funds from it.

Objectives:

- Implement and deploy a basic Ethereum smart contract in Solidity
- Understand basic smart contract vulnerabilities
- See how vulnerabilities can lead to stolen funds.

Part 1. Rock, Paper, Scissors

Rock, Paper Scissors is a simple game played between two players to determine which one is unlucky. Each player simulatenously makes a choice of rock, paper, or scissors, with the winner determined as follows: rock beats scissors, scissors beats paper, paper beats rock (???).
You will implement a simple Rock, Paper, Scissors (RPS) game in an Ethereum smart contract, to allow people to play this time-honored game in the modern era, for (testnet) Ether.

Part 1.1 Smart Contract

The game should work as follows. First, player one submits a wager with a certain amount of Ethereum to the contract, and a blinded commitment of their choice (rock, paper, or scissors). Player two submits an equal wager (if greater, the excess should be refunded, if less, the wager is invalid), and their commitment. Next, each player submits a transaction unblinding their commitment, revealing to the contract (and the world) what their choice was. The contract should determine the winner, and the winner should be rewarded with the wagers from both players, with the loser getting nothing. In the event of a tie, the players should be allowed to reclaim their own wager.

Your contract should have the following functions:

```
/* Given a choice (e.g. "rock", "paper" or "scissors")
   and a random/blinding string, returns a commitment
   of the pair. Note calls to this function occur offline
   (i.e. do not appear on the blockchain).
   If you leave the 'pure' keyword in the function signature,
   function calls to this won't publish to the blockchain.
*/
function encode_commitment(string memory choice, string memory rand)
  public pure returns (bytes32) { }

/* Accepts a commitment (generated via encode_commitment)
   and a wager of ethereum
*/
function play(bytes32 commitment) public payable { }

/* Once both players have commited (called play()),
   they reveal their choice and blinding string.
   This function verifies the commitment is correct
   and after both players submit, determines the winner.
*/
function reveal(string memory choice, string memory rand) public { }
```

```
/* After both players reveal, this allows the winner
   to claim their reward (both wagers).
   In the event of a tie, this function should let
   each player withdraw their initial wager.
*/
function withdraw() public { }
```

Think carefully about your design, and what you'll want to verify. Try to prevent funds from being locked into the contract by malicious players. Your game should reset after the winner has claimed their reward (or both parties in the case of a tie), allowing participants to play again if they choose.

You should use MyEtherWallet to test/play your contract against your project partner(s). Feel free to consult Rock, Paper, Scissors strategy to maximize your winning: `https://arxiv.org/pdf/1404.5199v1.pdf`.

Part 1.2 Reveal Timeouts

Note: **Required for graduate students** If your group has a graduate student in it, you must complete 1.2. Otherwise, this part is optional (but encouraged!).
For this part, your smart contract should properly handle a "reluctant revealer": imagine that Player One commits to "rock" and Player Two commits to "scissors". Then, Player One reveals their commitment, which lets Player Two know they will lose if they also reveal. To spite their opponent, Player Two never reveals that they picked "scissors", and so Player One can never collect their winnings, which are locked in the contract forever!
To handle this case, your contract should allow for one party to recover **all** the wagers if they have revealed without the other player revealing for a certain time (e.g. an hour). Alternatively, you can construct monetary incentives that encourage the loser to still reveal their commitment.

What to submit

1. A solidity file, named `rps.sol` that contains your smart contract source code.

2. A plaintext file, named `rps.txt` that contains the deployed address of your smart contract on the Ropsten test network.

Part 2. Vulnerable Contracts

In this part, you'll investigate how to steal Ethereum from vulnerable smart contracts. We have setup a vulnerable smart contract on the Ropsten testnet that you have permission to steal funds from. However, using this technique on other contracts you do not have the same permission for outside this class is a *crime*. Remember that just because you *can* do something technical, doesn't mean that you *should*!

We've deployed our contract to address `0x649A4bd91068077e1D7C9Ddf389a445234801794`. You can download its source from `https://ecen5033.org/static/vuln.sol`.
The contract has two functions: `deposit` and `withdraw` that let you send and receive money from the contract. On the surface, it appears that an address will only be able to withdraw what that address originally deposited. But this contract is vulnerable to a bug that lets you extract more if you're clever!

In this part, you'll write and use a contract that steals funds from the Vuln contract. Your goal is to make a contract that includes a `payable` function, that interacts with the Vuln contract to steal funds from it. Your contract should let you pay it a small amount (e.g. 0.1 ETH), and then later let you extract a greater amount (e.g. 0.2 ETH). If we look at the (internal) transactions between your contract and the Vuln contract, we should see that yours sends (deposits) less than it gets back (withdraws) from the Vuln contract.

Note : This is a class-wide contract, and only has so many testnet coins in it. As such, please limit your stealing to a humble amount, so that others can also enjoy the thrill of stealing as well. We suggest trying to deposit/steal small amounts such as 0.1 ETH (100 finney) at a time. You don't need to steal all the coins to prove that you can, but if you end up stealing too much, you can always `deposit()` it back.

Feel free to deploy your own copy of the Vuln contract (and fund it with your own testnet coins) to practice attacking before you try to attack the main Vuln contract.

What to submit

1. Your stealing contract, `attack.sol`, that you used to extract more funds than you deposited.
2. A plaintext file, `attack.txt`, that lists the deployed address of your contract.

Submission Checklist

Upload to **Dropbox** (`https://www.dropbox.com/request/SutPmxgypy2ASPDopqod`) a gzipped tarball (`.tar.gz`) named `project2.`*`identikey1`*`.`*`identikey2`*`.tar.gz`. You can make a tarball with

```
$ tar czf project2.identikeys.tar.gz ./preimage.py ./preimage.txt ./verify.py
```

The tarball should contain only the following files:

Part 1

A solidity file **`rps.sol`** implementing your Rock, Paper, Scissors game, and the **`rps.txt`** file that contains the address (starting with 0x..) of your deployed contract.

Part 2

Your stealing contract **`attack.sol`**, and a plaintext file `attack.txt` with its deployed address.