CS 5433 SP23: Homework 3

Instructor: Ari Juels
TAs: Andres Fabrega, Zhi Liu, Sishan Long
Due: 12 April 2023

Policies

Submit your written solutions and code to CMSX.

Integrity. For all assignments, you may make use of published materials, but you must acknowledge all sources, in accordance with the Cornell Code of Academic Integrity. Additionally, you must ensure that you understand the material you are submitting; you must be able to explain your solutions to the course instructor or TA if requested. You must complete this homework assignment *on your own.*

Disclaimer

In general, nearly all smart-contract technologies today are poorly (if at all) documented, are constantly changing, and suffer from broken or dead code or packages. That includes the ones for use in this assignment, which are fairly standard in the industry. **Please start the assignment early**. We do not guarantee responses from the TAs or instructors on errors in the assignment or such broken packages without at least 48 hours lead time.

Problem 1 - Tokens and Simple Smart Contracts

To get our hands dirty with the kinds of smart contracts we've discussed in class on Ethereum, we will create a simple token contract. The directory `ERC20` contains a test skeleton and instructions for installing/running dependencies in the file `README.md`, the version we use here should be $\geq$ 0.8.0. We also provide some Solidity code for a basic ERC20 token, as described in the specification here: `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`. Such tokens are the basis of the "ICO craze" that swept the cryptocurrency community and media some years ago: See, e.g., `https://techcrunch.com/2017/05/23/wtf-is-an-ico/`.

You may also find the Solidity documentation helpful, as the contracts we are writing will be Solidity code: `https://solidity.readthedocs.io/en/v0.8.12/` (and if you learn by example: `https://solidity.readthedocs.io/en/v0.8.12/solidity-by-example.html`). Solidity is extremely similar to Javascript, Java, or C/C++ syntax, and thus should be relatively familiar.

However, if you do find installing Solidity on your machine to be challenging, you can also use the online IDE Remix, which we will introduce later.

Your task is to enhance our basic ERC20 contract, provided in `ERC20.sol` as follows:

1. Change the token name to your NetID.

2. This particular ERC20 is meant to be pegged to the value of 1000 wei. A wei is the base unit of currency in the Ethereum system; see `https://eth-converter.com/extended-converter.html` for a conversion table. A new function `deposit` is included that permits a caller to generate and obtain ownership of a fresh token for 1000 wei. Given `msg.value` of x wei, the function generates $\tau = \lfloor x/1000 \rfloor$ tokens, assigns them to `msg.sender`, and refunds $x - 1000 \times \tau$ tokens to `msg.sender` (the remaining balance). Notice the use of the `msg.value` keyword, which provides the contract access to the current message value in wei.

   Your task now, is to add the corresponding `withdraw` function, allowing senders to redeem each token for the 1000 wei they deposited to obtain it. A placeholder is provided in the given file.

One way to test your contract is with the provided tests, which you can run by `python3 run_tests.py` (see `README.md` for dependency installation instructions). On top of that, you can also manually test your contract by trying a deposit and withdrawal on a live Ethereum test network (e.g., Sepolia). Once you have completed the contract, you must deploy it to the Sepolia network. Instructions for interacting with the testnet are below:

1. Install Metamask on Chrome. Follow the provided steps to set up a Metamask account. Switch Metamask to Sepolia testnet mode using the dropdown menu at the top right of the Metamask interface.

2. Get some Sepolia Ether. You can get Ether from the faucet: `https://sepoliafaucet.com/`. It might take a few seconds to minutes to see the updates to your Metamask account. Once Metamask on Sepolia shows your Ether balance, you are set.

   Sepolia is the Ethereum test network. You DO NOT need to pay money or purchase any actual Ether tokens.

3. Use `http://remix.ethereum.org/` to compile and deploy your contract.

   (a) To compile, select "Open Files" on the home page and select your ERC20.sol file. On the left hand side, you should see several tabs. Go to the tab that says "Solidity compiler" when you hover your mouse over it (this should be the third one down). Click "Compile ERC20.sol" to compile. If successful, you should see a green checkmark appear at the tab icon.

   (b) Next you will deploy your contract. Go to the tab that says "Deploy & run transactions" (fourth tab down). Switch the environment to "Injected Provider - Metamask" and select "MyToken - browser/ERC20.sol" in the drop down above the Deploy button. **Important**: make sure you have your "ERC20.sol" file open and not the *Home* tab open for deployment, or else you will get an error. Now click the Deploy button. A Metamask window should appear in your browser (if not, click the Metamask icon). Follow all on-screen prompts to confirm. After finishing, a link to the deployment transaction will appear in the Remix console (you can find this at the bottom of the Remix page). Once the transaction is mined, the contract address will be shown in that link in the "To" field. You can also find the contact address on the left side of Remix under "Deployed Contracts". To view your contract on the public testnet, you can enter the address of the contract at `https://sepolia.etherscan.io/` and see all relevant transactions.

Write your code in the provided **ERC20.sol** file and your deployed contract address (not the transaction hash of contract creation) in **ERC20_addr.txt** file. The txt file should contain one single line of a contract address such as `0x0BE207E608Af340c3B238901120A0fAa2bbc0E9C`. Don't put anything else in it.

Problem 2 - Gaming Contracts

Ethermon (`https://www.ethermon.io`) is a Pokémon-like game played via an Ethereum contract. Ethermon players catch or buy monsters, and then build their monsters' skills by means of training in a "gym" or by battling with other monsters. The outcome of a battle is determined by randomness derived from the blockchain using this function:

```
function getRandom(uint8 maxRan, uint8 index, address priAddress) constant public returns(
    uint8) {
    uint256 genNum = uint256(block.blockhash(block.number-1)) + uint256(priAddress);
    for (uint8 i = 0; i < index && i < 6; i ++) {
        genNum /= 256;
    }
    return uint8(genNum % maxRan);
}
```

Listing 1: Ethermon Randomness Source

`block.blockhash` is used to return the hash of the last block, which is converted to integer form. Some determinsitic post-processing then happens according to a randomness index and the address of e.g., the monster that is currently battling, ensuring uniqueness inside the contract.

We've created a simplified version of Ethermon called EthermonLite, packaged with the homework in `entropy/EthermonLite.sol`, that determines battle outcomes in a similar way, but without the additional address and index parameters. EthermonLite allows users to battle the house, and naturally biases battles heavily in favor of the house, represented as a monster called the Ogre. EthermonLite is running on the Sepolia testnet at this address: `0x04EAB7C83B2F45bDbE9DF44E337740bbdFe5efDE` and can be viewed on the Sepolia testnet explorer here:
`https://sepolia.etherscan.io/address/0x04EAB7C83B2F45bDbE9DF44E337740bbdFe5efDE`. The contract has been verified on etherscan for easier interaction from your browser.

Unfortunately, the method used for randomness generation in EthermonLite is vulnerable to manipulation.

```
uint dice = uint(blockhash(block.number-1));
dice = dice / 2; // Divide the dice by 2 to add obfuscation
if(dice % battleRatio == 0){
    monsters[challenger].wins += 1;
    monsters[Ogre].losses += 1;
    challengerWins = true;
}
```

Listing 2: EthermonLite Battle Code

Your task is to create a monster and hack EthermonLite so that your monster repeatedly defeats the Ogre.

You must submit:

1. The source code for a contract you used to accomplish your exploit. Use the template **Winning.sol** we provide in the `entropy` directory to get started.

2. The Ethereum address monsterAddress of a monster on the Sepolia network that has 2 or more wins and 0 losses on our deployed contract, and whose name (monsters[monsterAddress].name) is your NetID. An example of such a monster's address on-chain for the provided contract is `0x7e2F103fac21A17A8ddb98202E70A6f69A7132dB`.

Hints:

- Instead of the obvious way of having an account own your monster, have a *contract* own your monster. Remember, each contract has an address and can make calls to other contracts, hold tokens, contain a constructor that performs setup, etc.! Launch your own local instance of EthermonLite in Solidity and play with these monster-owning-contracts, but remember, Remix will not return real blockhashes so you must do your final testing on-chain. When doing your on-chain transactions, make sure to use enough gas; we recommend 100,000 gas for your outer contract calls.

- Check out the transactions that made the working example we provided work; you could always try to reverse engineer their EVM bytecode, but it will probably be easier to write your own code. If you do choose the reverse-engineering route, please try to understand *why* the code works and what this means for why randomness in smart contracts is difficult.

Write your code in **Winning.sol** file, and your monster's address in a new **Winning_addr.txt** file. Again, don't put anything else in the text file.

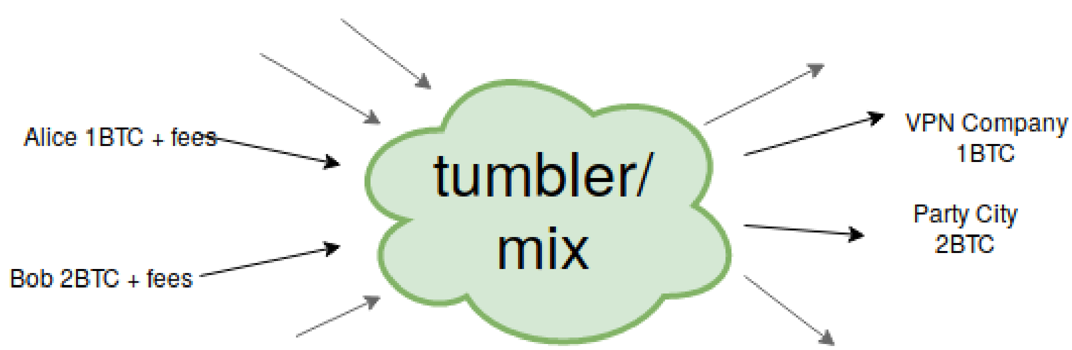
Problem 3 - What Anonymity?

As observed in class, Bitcoin does not provide full anonymity, but instead offers a weaker form of privacy. This limitation is the reason for use of fresh change addresses in Bitcoin transactions, and has also given rise to what are called “mixers” or “tumblers.” (See, e.g., `https://en.wikipedia.org/wiki/Cryptocurrency_tumbler`.)

A mix is a service that ingests a set of outputs (BTC) from a set of addresses $A_1, ..., A_n$ and redistributes them to a fresh set of addresses $B_1, ..., B_m$. An observer of a mix's on-chain transactions does not explicitly learn a correspondence between incoming and outgoing addresses.

Mixes can in principle enhance user privacy. For example, suppose that Alice, Bob, and Charlie respectively own $\texttt{addr}_A, \texttt{addr}_B$, and $\texttt{addr}_C$, each with 1 BTC, and this ownership is known to an adversary. If Alice, Bob, and Charlie want to conceal their ownership of BTC, they can send 1 BTC into a mix from their respective addresses $\texttt{addr}_A, \texttt{addr}_B$, and $\texttt{addr}_C$, and have the mix send 1 BTC each to $\texttt{addr}_D$, $\texttt{addr}_E$, and $\texttt{addr}_F$, fresh addresses also controlled respectively by Alice, Bob, and Charlie. If the order of the outputs to these three addresses is randomly permuted in the UTXOs created by the mix, the adversary will be unable to tell if $\texttt{addr}_D$ belongs to Alice, Bob, or Charlie. Consequently, in observing a transaction from $\texttt{addr}_D$, the adversary will be unable to tell which of the three players spent the money. (Note: We are disregarding transaction fees in the example here.)

Tumblers are a type of mix, as is CoinJoin, where users perform a series of transactions together with their coins in a decentralized protocol. Unlike CoinJoin, for most tumblers, users send all money to a centralized service which internally mixes their money together. All users are paid out with a random UTXO held by the mixer, that comes from some other user, breaking the link between the funds on-chain to all but the operator of the tumbler. The operation of both is roughly summarized by the below diagram, which shows Alice mixing a 1BTC payment to her VPN company with Bob's 2BTC Party City payment. Whether a tumbler or a mix is used, the correspondence between inputs and outputs is hidden. In the case of a tumbler, some delay may also be added between the two transactions to prevent timing attacks, and some random fee is charged to increase anonymity. Mixes also charge fees that can potentially be randomized.



Law enforcement and tax collection agencies have employed companies such as Chainalysis (`https://www.chainalysis.com/`) to identify illegal activities such as tax evasion. Mixes and tumblers can make their task more difficult.

In practice, however, mixing or tumbling can offer weaker than ideal privacy. First, as above, players may send unequal amounts into a mix or tumbler. Second, a mix or tumbler may be used to conceal payments, rather than just impart greater privacy to an existing set of players. This usage may constrain the choice of output values emitted by the mix. For example, if Alice is paying a service exactly 1 BTC, then a subset of outputs must sum to 1 BTC. Often goods and services involve payments in round amounts (e.g., .1 BTC or BTC equivalent to \$100 USD), making it easier to correlate inputs and outputs.

Your task in this exercise is to partially deanonymize a collection of real tumbler operations observed in Bitcoin.

Consider the following input addresses to a series of tumblers:

- 1MVXpgczazLvbtS8Nfp9v3Qpj4d8pUNXQM (Grams Helix - grams7enufi7jmdl.onion/helix)
- 135g5Es7VXvbaAkwzguv7q7xaSSTifav5H (Bitcoin Fog - foggeddriztrcar2.onion)
- 1GcZjZnfQUCs9L9RoAFLdd8YET2WQWrDAz (CoinCloud - coincloud25txgdf.onion)
- 1KGhtebk4Nr2zZSn2NaFepeNF6KyjxpPJZ (PenguinMixer - penguinsmbshtgmf.onion)

and the following outputs:

- 18RwKzXtL5YGvFwa9BHrPRvqXLkdYWsGfp
- 1MTbp4bFftessrbTTpM5SC5Ap1iKaMHrM7
- 1BCaztysy2paguXjuC8c652vckNMks69ce
- 13MUZ1Qk36LqExdcSRDZCxNRP1pcz1b5mT

You can use blockchain.com to view the transactions on a given address; for example `https://www.blockchain.com/btc/address/13MUZ1Qk36LqExdcSRDZCxNRP1pcz1b5mT`.

In **solutions.pdf**, deanonymize the mixers by listing pairs of input/outputs that are part of the same mix operation. Briefly comment on how you were able to deanonymize these transactions, and what this implies about mixing Bitcoin on-chain.

Submission Instructions

Directory Structure:

```
hw3-<NetID>
├── p1
│   ├── ERC20.sol
│   └── ERC20_addr.txt
├── p2
│   ├── Winning.sol
│   └── Winning_addr.txt
└── p3
    └── solutions.pdf
```

Evaluation

To help us tune future homeworks in the class, please answer the following in your **p3/solutions.pdf**:

- Did you find the homework easy, appropriately difficult, or too difficult?
- How many hours total (excluding breaks :)) were spent on the completion of this assignment?
- Did you feel there was too much coding, the appropriate amount of coding, or not enough coding?

Any other feedback on the homework or class logistics are appreciated!