CS251: Cryptocurrencies and Blockchain Technologies **Fall 2021**

Programming Project #3

Due: 11:59pm on **Wed., Nov. 10, 2021**
Submit via Gradescope code: **D5GKRX**

For this project, you will be working on a web client and two Solidity contracts to implement a decentralized cryptocurrency exchange. By the end of the project, your exchange will have much of the functionality possessed by full fledged decentralized exchanges such as Uniswap. Additionally, you will create your own ERC20 token, which you will then trade over your exchange.

Throughout this project, make sure to write code that is resilient against adversarial attacks. Remember that attackers can call your smart contracts with arbitrary inputs, not just through the provided user interface. As such, make liberal use of `require` statements to test any assumptions baked into your code. There's real fake money at stake here, so it should not be possible for an enterprising attacker to rip off your exchange.

For most students, this project will likely represent their most comprehensive project implemented so far in Solidity. As such, we encourage you to start early and ask questions. We will guide you through implementing a decentralized exchange in several phases, and we believe your end product will give you something to be very proud of! Let's get started.

1 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 the `exchange.sol` and `exchange.js` files. You may add helper functions to these two files if needed.

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 0.8.0+.

5. Under the "File explorers" tab on Remix, click the upload icon underneath "default_workspace." Upload the entire `contracts`, `interfaces`, and `libraries` directories from the starter code. After uploading, you should see the `contracts/exchange.sol`, `contracts/token.sol`, `libraries/safe_math.sol`, and `interfaces/erc20_interface.sol` files in your Remix "File explorers" tab.

6. Open `index.html` in your browser (works best in Chrome) to access the user interface through which you can test your exchange. On this page, you can pick an address and add liquidity, remove liquidity, and swap your token for ETH and vice-versa.

7. 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).

8. Implement code for the requirements outlined below. When you have your contract, compile and deploy it, and then **update the contract address and ABI** at the top of `exchange.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.

2 Create and Deploy Your Own Token

In the first part of this project, you will create and deploy your own ERC20 token in Remix. ERC20, as mentioned in class, is a standard for implementing fungible tokens. Luckily for us, much of the code for ERC20 standard has already been written and is open source. In this project, we used the ERC20 code from the OpenZeppelin project, which has been included with light modifications in the `token.sol` file. Read through the starter code and make sure you understand how the functions under the header "STANDARD ERC-20 IMPLEMENTATION" work.

Once you've read through the starter code, complete the following steps:

1. Come up with a fun (but appropriate!) name for your token. Any name is fine, so feel free to have fun and come up with something creative. Replace the name of your contract (currently `Token`) with the name of your token. In addition, set the public string `name` to be the name of your token. Replace the type of the `private token` variable in `exchange.sol` with the name of your token contract. Lastly, update the `token_name` variable at the top of `exchange.js`.

2. Decide on a short symbol for your token (e.g. ETH, instead of Ethereum). Set the public string `symbol` to be your symbol, and update the `token_symbol` variable at the top of `exchange.js`.

3. Take note of (but don't change) the `decimals` variable. This variable exists because Solidity does not have floating point types, so fractional tokens can't be represented. As such, instead of representing a fractional token as a decimal, **we represent the tokens in units of $10^{-\texttt{decimals}}$**. For example, if `decimals` is 18, then calling `transfer(receiver, 1)` will transfer $1 * 10^{-18}$ tokens to the receiver. This is analagous to using Wei as a subunit of Ethereum, or Satoshis as a subunit of Bitcoin.

4. Implement the `mint(uint amount)` and `disable_mint()` functions. `mint` is a public function that creates new tokens, adjusts the `_totalSupply` variable, and sends those tokens to the administrator of the contract. `disable_mint` makes it such that calling `mint` will never succeed again. As such, your `mint` implementation must fail if `disable_mint` has been called. Also note the `AdminOnly` modifier on both functions – this makes it such that only the contract administrator can call either one. The `mint`/ `disable_mint` paradigm is a simple way of

initially generating tokens and guaranteeing that the total supply will remain constant after the supply of the token has been created.

5. Deploy your token contract. Copy the address and ABI of the token contract to the `token_address` and `token_abi` variables in `exchange.js`.

6. Lastly, copy the address of the token contract to the `tokenAddr` variable in `exchange.sol`. **Every time you redeploy your token contract, you must repeat this step.**

After completing the above steps, you should have your own token all deployed on Remix!

You may now try deploying your exchange contract on Remix as well. Copy the address and ABI of the token contract to the `exchange_address` and `exchange_abi` variables in `exchange.js`. All the functionality except initial pool setup (which we implemented for you) will be missing. That said, if you have completed Section 2 and the deployment process properly, you should see no errors occur in your browser console and should see a 1-to-1 exchange rate between ETH and your token. There should be 1 ETH and 1 of your tokens in under the "Current Liquidity" display.

3 Setting Up Your Basic Exchange

In this part of the assignment, you will implement the basic functionality of your cryptocurrency exchange. Our exchange is modeled after Uniswap V1, and we have linked the whitepaper on those words and in the code for your reference. Your exchange will only allow for swaps between two tokens: your token created in Section 2 and test ETH. The changes in this section will primarily affect two files: `exchange.js` and `exchange.sol`. Familiarize yourself with the starter code for those files. Remember to use the `SafeMath` library for all arithmetic operations.

A decentralized exchange consist of two types of participants: liquidity providers and traders. For a given exchange pool between two tokens, liquidity providers provide some equal value amount of both types of liquidity (in your case, ETH and your own token). When traders swap between the two currencies, they will add some amount of one currency to the liquidity pool, and will be sent an equal value of the other currency from the pool. The exchange rate between the two currencies is determined by some price curve. In our case, we are going to be using the constant product formula:

> Let x be the amount of currency A that is in the liquidity pool, and let y be the amount of currency B. Let k be some constant. After every swap, it must be true that
>
> $$x * y = k.$$

This means that, during each swap, the exchange must send out the correct amount of the swapped-to currency such that the pool always remains on the constant product curve. The price of currency B in terms of currency A can the be calculated as x/y, whereas the price of currency A in terms of currency B can be calculated as y/x. This means that every swap will change the price of the cryptocurrency; this makes sense, as each swap is an indication of demand for a given currency. The effects of each swap on the price of the currencies will be relevant in section 4.

When a liquidity provider adds liquidity, they must provide equal values of currency A and B, as determined by the current exchange rate. Note then that adding liquidity will not change the exchange rate between the currencies, but will increase the value of the constant k. Similarly, when a liquidity provider goes to withdraw their liquidity, they must withdraw equal values of currency A and currency B. Therefore, k will decrease but the exchange rate will remain the same.

This has another notable consequence: since a liquidity provider can only withdraw equal values of each currency, they are not actually entitled to withdraw their exact initial investment (in terms of quantity of each token). Rather, providing liquidity is analagous to owning a percentage share of the liquidity pool, which the provider is then entitled to withdraw at a later time. A liquidity provider who provided 10 percent currency A and currency B is entitled to withdraw 10 percent of each of the reserves for those currencies (assuming their percentage was not diluted by other providers), even if 10 percent does not line up precisely with the quantities of each token they provided. In this way, liquidity providers can suffer impermanent loss, if the value of what they are entitled to withdraw is less than the value that they initially invested. If you are unfamiliar with impermanent loss, review the lecture and section on decentralized exchanges.

With the above in mind, you will now implement the basic functionality of your exchange. Also be aware that we take care of initializing the pool for you by implementing and calling your `createPool` function. Initially, the first address in the list of addresses on `index.html` is the sole liquidity provider of all ETH and tokens. In order for other addresses to obtain tokens and/or provide liquidity, they must first swap for tokens on the exchange. In `exchange.sol`, implement the following functions:

- `function priceToken() public view returns (uint)`:
  Determine the price per token in ETH based on the current exchange rate. Be sure to use the `SafeMath` library when performing all arithmetic operations.

- `function priceETH() public view returns(uint)`:
  Determine the price per ETH in tokens based on the current exchange rate.

- `function addLiquidity() external payable`:
  Add liquidity to the pool if the provider possesses sufficient ETH and tokens (otherwise the transaction should fail). The caller will send ETH to the contract as the `msg.value`. The contract should transfer the equivalent amount of tokens from the sender's address to the contract (using the token's `transfer` or `transferFrom` method) and update the exchange state accordingly. The transaction must fail if the provider's funds are insufficient.

- `function removeLiquidity(uint amountETH) public payable`:
  Remove liquidity from the pool (if the provider is entitled to remove given amount of liquidity) and update the exchange state accordingly. The amount of liquidity to be removed is specified by `amountETH`, and the provider also receives an equal value of your token.

- `function removeAllLiquidity() external payable`:
  Remove the maximum amount of liquidity that the sender is allowed to remove and update the exchange state accordingly.

- `function swapTokensForETH(uint amountTokens) external payable`:
  Swap the given amount of tokens for the equivalent value of ETH and update the exchange state accordingly. If the provider does not have sufficient tokens for the swap, the transaction should fail. Additionally, if completing the swap would completely remove all ETH from the pool, the transaction should fail to avoid having zero ETH and (therefore) an undefined exchange rate.

- `function swapETHForTokens() external payable`:
  Swap the given amount of ETH for the equivalent value in your token and update the exchange state accordingly. The number of ETH provided to the contract is specificed in `msg.value`. If completing the swap would completely remove all tokens from the pool, the transaction should fail to avoid having zero tokens and (therefore) an undefined exchange rate.

In each of the above functions, be sure that you are adjusting `token_reserves`, `eth_reserves`, and/or `k` in the correct way such that the exchange is always remaining on the constant product curve described above. Additionally, be sure that functions fail when the caller does not possess sufficient funds. Finally, remember to set `address tokenAddr` to be your deployed token contract's address.

In addition to implementing the smart contract functions in Solidity, implement each of the corresponding functions in `exchange.js`. In most cases, this will just consist of calling your Solidity function with the correct parameters. The (already implemented) `approve` method in `token_contract.sol` will also be necessary in some cases. **For now, you can ignore the `maxSlippagePct` parameter in each JavaScript function**. Make sure to come up with a fun name for your exchange, which you can set in the `exchange_name` variable.

Once you've fully implemented your smart contract and the corresponding JavaScript code, deploy the contract, copy your ABI to the top of your `exchange.js` file, and copy the contract address to the `exchange_address` variable in `exchange.js`. If you reload `index.html`, you should now be able to provide liquidity, remove liquidity, and perform swaps.

4 Handling Slippage

There is a significant issue with our exchange as we implemented it in Section 3, as it does not account for ”slippage”. Recall that with every swap on a decentralized exchange, the price of each asset will shift slightly. Since many users may be trying to swap currency at once on a decentralized exchange, there may be a shift in the exchange rate between the submission of a swap transaction and the actual processing of that transaction. This shift in the exchange rate between the exchange's quote price and actual price is called ”slippage.” Slippage is of particular concern while trading volatile assets. For example, if a user submits a swap transaction to swap some amount of currency A for currency B, and then the price of currency B dramatically increases from the quote price, the user might not actually wish to complete the swap transaction.

Additionally, not sufficiently handling opens users up to a type of attack known as a sandwich attack. A sandwich attack works as follows:

1. Alice submits a swap transaction to convert some large amount of currency A into currency B.

2. An adversary sees Alice's transaction and front-runs it with a very large purchase of currency B, thus raising the price of asset B.

3. Alice buys currency B at the new higher price, even further raising the price of currency B.

4. The attacker then immediately sells all their newly acquired currency B at the higher price, making a quick profit.

Vulnerability to sandwich attacks is bad for users of a decentralized exchange, as users consistently pay higher exchange rates than the true asset value. As such, it is important that we upgrade our exchange to properly handle slippage and defend against sandwich attacks.

The most common defense against sandwich attacks is to allow users to set some maximum slippage while submitting the transaction. This parameter, typically a percentage, will cause the transaction to fail if the price of the assets has changed by more than the maximum allowed slippage. This limits the damage that can be done by sandwich attacks and protects users who are swapping volatile assets.

To implement a maximum slippage requirement, perform the following steps:

1. In `exchange.sol`, update each of your `swap` functions to take in a `uint max_exchange_rate` parameter. You may also pass in other parameters if needed for your design. While swapping, the swap should fail if the current price of the new asset (i.e. the asset the user is swapping to) has increased to more than the maximum exchange rate. Note that the price of the asset decreasing is good for the user, so we don't have to fail in that case. Be careful in your calculations, as since `max_exchange_rate` is an `uint`, you will need to divide by 100 at appropriate times.

2. Update each of your `addLiquidity` and `removeLiquidity` functions to take in `uint max_exchange_rate` and `uint min_exchange_rate` parameters. You may also pass in other parameters if needed for your design. While providing liquidity, the transaction should fail if the current price of the new asset (i.e. the asset the user is swapping to) has increased to more than the maximum exchange rate or fallen below the minimum exchange rate. Sudden price shifts in either direction can subject providers to impermanent loss before they deposit their liquidity, so thus we want the liquidity providers to specify a maximum and minimum exchange rate.

3. Now update your `exchange.js` file to communicate with the contract about the max/min exchange rates. The `maxSlippagePct` parameter is provided, which represents the maximum allowable percent price change before the transaction should fail. (The parameter is passed as an int, not as a float- i.e. 4% is passed as 4, not 0.04). This parameter can be used in each of the JavaScript functions to calculate the correct values for `max_exchange_rate` and/or `min_exchange_rate`, which can then be passed to the contract. The `getPoolState` function that we provide to you may be useful here.

As always, after updating your contract make sure to recompile, redeploy, and copy the new ABI and contract address to the variable at the top of your `exchange.js` file.

5 Rewarding Liquidity Providers

After completing the above sections, you now have a working exchange that allows users to limit the amount of slippage they wish to tolerate! There is one more big issue, however. We have discussed several times how liquidity providers are taking on risk in the form of impermanent loss. That is, the value of their liquidity stake may decrease if the price of either asset changes. In practice, since many cryptocurrencies are quite volatile, this is a level of risk that no liquidity provider would be willing to take on for free.

As such, we need to incentivize liquidity providers to give liquidity to the pool. In real world exchanges, liquidity providers are incentivized to provide liquidity because they receive a small fee from every swap transaction. These fees are automatically reinvested into the liquidity pool on behalf of each liquidity provider. When a provider goes to withdraw their liquidity, the amount that they are entitled to withdraw includes all fees they have been awarded since providing their liquidity.

You will now implement the same fee reward scheme for liquidity providers.

Note: This section will require you to design your own fee collection/distribution system. We provide the requirements for said system below, and we will accept any design that fulfills the below requirements.

Liquidity Rewards Requirements:

1. Your pool must charge the person performing the swap some nonzero percent fee on every swap transaction.[1] Fill in the `swap_fee_numerator` and `swap_fee_denominator` with a fraction representing your chosen percentage. For example, for 5%, set `swap_fee_numerator` to 5 and `swap_fee_denominator` to 100. This is needed as Solidity does not have floats.

2. When a swap occurs, the value of tokens or ETH sent to the trader should be equal to $(1 - p)$ times the value of the assets they are swapping, where p is the percent fee taken for liquidity providers. For example, if the fee is 1% and a user is swapping 100 ETH for your tokens, they should only be sent the equivalent of 99 ETH.

3. When a fee is taken during a swap, it should be distributed to the liquidity providers such that each provider should later be able to withdraw their original stake plus fees. Fees should be distributed proportionally based on providers' fractional share of the liquidity pool at the time that the swap took place. It would be incorrect, for example, if a liquidity provider who provided half of all liquidity in the pool at time t was allowed to withdraw half of all fees taken prior to time t. Liquidity providers should not have to take any additional steps to claim their fees beyond calling removeLiquidity. Additionally, liquidity rewards should *not* be sent out of the exchange to the providers each time a swap takes place, since doing so would be prohibitively expensive in practice.

4. Pending their withdrawal, all fees should be automatically reinvested into the pool on behalf of each liquidity provider. That said, it is important to make sure that swaps never violate the constant product curve of $x * y = k$. In the example from (2), if only 99 ETH's worth of tokens is leaving the pool, then we cannot immediately add 100 ETH to the pool or else k will change. Instead, 99 ETH should be added to the pool immediately, whereas the 1 ETH taken for liquidity provider rewards should be reinvested into the pool *only* when we can do so without violating the constant product curve. As such, this fee reinvestment does not have to happen during every single swap function, but it should happen shortly after it becomes possible to do so without violating $x * y = k$.

5. While deciding between different design options, we encourage you to opt for the solution that minimizes gas costs. We will not grade strictly on gas usage; however, you will be required to justify your design decisions in the design doc in Section 6.

[1] For reference, the default fee on Uniswap is 0.3%, whereas centralized exchanges typically charge around 1-4% to swap currencies.

After designing and implementing the above section, you should have a fully working exchange! Congratulations! Test your functions using the provided UI in `index.html`, or write testing code in JavaScript. Implementing this project represents a very impressive achievement, so give yourself a pat on the back. In fact, with some security modifications, you can deploy both your token and your exchange onto the Ethereum mainnet, and thus have an exchange you can call your own!

6 Note on Solidity and Javascript Decimals

Unlike most programming languages, Solidity does not support floating point arithmetic. Thus, all ERC-20 tokens keep track of a *decimals* variable, which indicates how many decimals to the left the numbers should be interpreted as. For example, ether uses 18 decimal points, so 1 ETH would be represented as 10^{18} in the contract. Similarly, 1 wei = 10^{-18} ETH, so 1 wei is represented as just 1. Unfortunately, Javascript also has a limit to how large integers can be: Numbers.MAX_INT = $9 * 10^{15}$. To balance out between the two, in our exchange, we initialize the pool to have $10^{1}0$ tokens, which means that the pool starts off with 10^{-8} ETH and 10^{-8} of your tokens.

The consequence of Solidity not supporting float operations is that all decimal numbers would be truncated. For example, 5 / 2 = 2. This is why the decimals variable is needed, as representing 5 ETH as $5 * 10^{18}$ would lead to $5 * 10^{18}$ / 2 = $25 * 10^{16}$, and since we know there are 18 decimal places, we can see that this corresponds to 2.5 ETH, as desired. However, in some cases, underflow can still occur, especially if the numerator is smaller than the denominator in the calculations and thus leading to 0. In this case, you would need to be careful with the order of operations, such as multiplying or adding first before dividing. In general, it is a good idea to delay division until as late as possible to prevent encountering this rounding error.

In this project, we will not be testing you on overflow. Thus, when testing our your exchange, we will choose values that won't have your exchange exceed Javascript's INT_MAX. However, we will be checking your implementation to make sure you have accounted for underflow in your exchange.

7 Design Document

Please fill in `DesignDoc.txt` with your answers to the following questions:

1. Explain why adding and removing liquidity to your exchange does not change the exchange rate.
2. Explain your scheme for rewarding liquidity providers and justify the design decisions you made. How does it satisfy requirements (2)-(4) from Section 5?
3. Describe at least one method you used to minimize your exchange contract's gas usage. Why was this method effective?
4. Optional Feedback
   (a) How much time did you spend on the assignment?
   (b) What is one thing that would have been useful to know before starting the assignment?

(c) If you could change one with about this assignment, what would you change?

(d) Please feel free to include any other feedback you may have.

8 Submission

When you are ready to submit, please upload a .zip file of your entire project to Gradescope. Be sure to include the following files in your project zip: `exchange.js`, `exchange.sol`, `token_contract.sol`, and `DesignDoc.txt`. If you added or changed any other files, please be sure to include those files as well.

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.