COMP6451 Assignment 2: Ethereum Programming

By Dumidu Thenuwara (z5205998)

Data Model

For the requirements given to us I opted to use a larger "University" contract as the primary entry point into the university system with several smaller libraries that help to keep the size of the primary contract down.

Contract/Library	Description
University.sol	Main entry point to the university system. Represents one session. Manages the roles of all the users and ensures state changing functions are protected and inputs are validated. - Roles are stored as a mapping(address => Role) where role can be one of {Unkown, Student, Admin, Lecturer} chief role is implied from the address stored as the owner of the contract - Stores students in a session as a mapping(address => Student) where the student struct stores how much UOC, the courses they have been accepted into and their currently active bids Delegates management of admission tokens and courses to AdmissionTokenLib and BiddableCoursesLib respectively. Utilizes the StudentRecord passed into the constructor to confirm validity for student to join a course. Utilizes ECDSA library from @openzeppelin library [1] to handle extracting the signer from a signature.
AdmissionTokenLib.sol	Handles all functionality related to admission tokens. - Storing the admission token balance of all users as a mapping(address => balance) - Minting tokens to assign to students - Burning tokens used on courses - Transferring tokens between users
BiddableCoursesLib.sol	Handles the collection of courses in a session as well as handling bidding on said courses. - Stores a struct BiddableCourses which contains a list of all course codes that have been added as well as a mapping(bytes8 => BiddableCourse) where BiddableCourse contains further details about the quota, UOC, lecturer, prerequisites, students that have been accepted and bids on the course - Delegates granular handling of bids to BidListLib.sol
BidListLib.sol	Handles the adding, changing and removal of bids from a course. - Stores bids as Linked List created from a mapping(address => Bid) where a Bid stores the amount of tokens placed on this course by the specified address as well as the "next" address as is commonly done in Linked Lists. This approach was chosen to lower the cost of inserting bids into an array in sorted order as an array would incur the cost of shifting numbers or repeatedly searching for the highest bid.
StudentRecord.sol	This contract is what stores the completion of prerequisites for each student in a mapping(bytes8 => mapping(address => bool)) where bytes8 is the course code. This contract could even be used between sessions to save admins incurring the cost of setting the completion of prerequisites for all students each session. - Very simple contract that just exposes a "hasPassed" function and a "pass" function to set and get the completion status of a prerequisite.

__Requirements__

The prototype should be able to manage enrolment for one session, with an arbitrary number of bidding rounds.

University.enroll(bytes32 hash, bytes memory signature) public hasNoRole

- Allows students to enroll in the university by providing a signed message from an admin that contains the address of the University contract and the public address of the student.
- It validates the signed message and assigns the Student role to the message sender.

University.startBiddingRound(uint roundTimeInSeconds) public isInitialized onlyAdmin

- Allows admins to start an arbitrary number of rounds, provided that a round is not currently active.

University.closeBidding() public onlyAdmin

- Allows admin to close the current bidding round provided that the current block timestamp is passed roundTimeInSeconds.
- Processes all of the bids made in the round and assigns student to their appropriate courses.

People using the system should be identified by means of an anonymous Ethereum address.
The system should support the following user roles: (many) students, (many) university administrators, and (one) university chief operating officer.
Only the chief operating officer has the ability to control which users act in the roles of the university administrators.

University.addAdmins(address[] memory addresses) public onlyChief

- As seen in previous section with **enroll** and the two functions directly above, only Ethereum addresses are stored, and the chief operating officer is initially the Ethereum address that created the contract.
- All role assignments are accounted for by these functions and the assumption that the creator is initially the chief operating officer.
- Only the chief can add admins as denoted by the **onlyChief** modifier

Chief may authorize users to act in admin roles, and revoke a user's authority to act in an admin role. They may also transfer their chief operating officer powers to another user.

University.removeAdmin(address addr) public onlyChief

- Allows chief to remove admin

University.transferChief(address addr) public onlyChief

- Allows chief to transfer their role to another address
- Requires that the new address does not have a role already

Only the chief operating officer should have the ability to

- *set the amount of fees (in Wei) per unit of credit*
- *transfer money out of the admissions system.*

University.init(uint8 _maxUOC, uint248 _feePerUOC) public onlyChief

- Allows chief to initialize a University contract with the fee per UOC and the max UOC that any student can take in the session.
- Only the chief operating officer can call this because of the **onlyChief** modifier

University.withdraw() public onlyChief

- Allows only the chief to withdraw the balance of the contract

Only university administrators should have the ability to:
- *admit a student to the university*
- *create a new course, setting its admissions quota, and number of UOC*
- *change the quota on a course*
- *set a bidding round deadline*

- Admission is done via **enroll** above. It checks that the signer has the admin role in the system thus ensuring that only admins can admit a student. Signing approach was taken to transfer the gas cost of enrolling to the many students.

University. createCourse(
bytes8 code,
uint quota,
uint8 uoc,
address lecturer,
bytes8[] memory prereqs
) public onlyAdmin

- Allows only admin (**onlyAdmin** modifier) to create a course by providing an 8-byte course code (e.g. 'COMP6451'), a quota, the number of UOC the course is worth, a lecturer's public address and an array of prerequisites which are also denote by their 8-byte course code.

University.setQuota(bytes8 code, uint quota) public onlyAdmin

- Allows an admin to change a course quota. Reverts if a bidding round is currently active. Also reverts if the new quota provided is less than the number of students that have already been accepted to the course.

- **startBiddingRound** mentioned above takes a number of seconds that the bidding round is valid for which it uses to calculate the timestamp after which attempts to bid or change bids are reverted. The end timestamp is compared against the current block timestamp.

Students should be able to do the following:
- *pay fees (in Wei) for a given number of units of credit*
- *bid some of their admission tokens for admission to a course*
- *change their bid before the round deadline*
- *transfer some of their tokens to another student*

University.payFees(uint8 numUOC) public payable isInitialized onlyStudent

- Allows a student to pay ether in wei for a specified number of UOC.
- Checks that msg.value is greater than or equal to numUOC * feePerUOC.
- Checks that numUOC is less than or equal to maxUOC.
- Mints appropriate number of admission tokens and assigns them to the student.
- Records the amount of UOC this student has paid for.

University.makeBid(bytes8 code, uint amount) public biddingIsOpen onlyStudent isCourse(code)

- Allows a student to bid on course
- Checks that the course requested will not exceed the number of UOC the student paid for
- Places bid on a course. Reverts if the user has already bid on the course. Accepts bid of 0 tokens.

University.changeBid(bytes8 code, uint amount) public biddingIsOpen onlyStudent isCourse(code)

- Allows a student to change their bid on a course
- Checks that token balance + original bid is greater than or equal to the new bid.
- Reverts if user has not made bid yet.

University.receiveTransfer(
bytes32 hash,
bytes memory signature,
uint amount,
uint nonce
) public payable onlyStudent

- Allows a buyer of tokens to receive tokens from another student.
- Details below in related requirement

Only students who have been admitted to the university may pay student fees and receive admission tokens.

- **payFees** has **onlyStudent** modifier which ensures that the caller of the function is enrolled as a student

Students should not be able to bid more tokens in any round than they own at the time of that round.

- **makeBid** and **changeBid** ensures that the bid requested is less than or equal to the token balance of the student that called the function via a require statement.

In any round, students may not bid for courses totalling more units of credit than they have paid for.

- **makeBid** has additional require statements to check that the student is not trying to bid on more UOC than they have paid for.

When a student is admitted to a course at the end of a round, the tokens that they have bid for the course in that round are destroyed

- **closeBidding** ensures that tokens in bids by users that are accepted into a course are burned.

Students should not be able to enrol in a larger number of units of credit than they have paid for

- when **makeBid** checks that a user is not exceeding the number of UOC they have paid for it is also factoring the UOC of courses that the user has already been accepted into.

It should not be possible for a student to transfer tokens without the university receiving the transfer payment.

- **receiveTransfer** has a require statement to assert that msg.value is greater than or equal to one tenth of the value of the tokens

Your design should facilitate the secure sale of enrolment tokens for cryptocurrency, so that students cannot cheat each other when selling some of their tokens. However, the university does not wish to have legal liability for such transactions and does not want to be a party to the actual trans-fer of money when a student sells some of their tokens to another student. Money paid by a student for tokens that they are buying should not passthrough any smart contract controlled by the university. The university merely wishes to ensure that they will collect fees for the transfer.

- In **receiveTransfer** above, transfer of ether between parties is handled outside of university owned contracts. There are ways to ensure that neither party has financial incentive to cheat the other via a Safe Remote Purchase model as described in the solidity docs [2]. Therefore, it can be treated like the buyer is buying a signed message from the seller of the tokens.
    1. The buyer sends the seller the address of the uni contract, their own (buyer) address, an amount of tokens to buy and a nonce
    2. Seller agrees and places twice the value of the tokens into the Safe Remote Purchase contract.
    3. Buyer confirms purchase and also places twice the value of the tokens into the Safe Remote Purchase contract. The funds are now locked. Thus 4 * the value of the tokens are in the contract.
    4. Seller sends the signed message back to the buyer

5. Buyer verifies that the message has not been tampered with and has been signed by the seller. They then call **receiveTransfer** from the University contract.
6. If the transfer is successful, the buyer confirms to the Safe Remote Purchase contract that they have received the tokens from the seller. The buyer is refunded the value of the tokens thus losing just the value of the tokens overall.
7. The funds are unlocked and the seller can claim 3 * the value of the tokens. Thus gaining just the value of the tokens overall.

- Nonce is recorded and cannot be used again to prevent double calls with the same signed message.

The system should be cost effective for the university to run. To the largest extent possible, costs for running of the system should be borne by students rather than by the university.

- **enroll** is not called by the university. Students are able to enrol themselves by providing a message signed by an admin containing the address of the contract and the public address of the student. This saves the Uni a lot in gas costs and transaction fees as there are thousands of student that need to be added.

There is an additional role of lecturer in the system (also as an Ethereum Address). The university administrators may authorize Ethereum addresses for this role.

University.addLecturer(address addr) public onlyAdmin

- Allow admin to add lecturer to the system

Course data should include a lecturer.

- As seen, **createCourse** accepts the address of a lecturer. This address is verified to ensure that they are indeed a lecturer. This address will be stored with a course.

It should be possible to represent which students have met the prerequi-sites for each course. University administrators enter this data.

StudentRecord.hasPassed(bytes8 courseCode, address student) public view returns (bool)

- Allows anyone to check whether or not a student has passed a course.

StudentRecord.pass(bytes8 courseCode, address student) public onlyAdmin

- Allows admin to set the passed status of a student for a course.

To bid to enrol in a course, a student should either have the prerequisites, or permission of the lecturer of the course.

- The University contract is passed the StudentRecord contract as an argument to its constructor. **makeBid** will cross-check the prerequisites of the course being bid on with the courses that a user has passed and revert if the user has not passed the prerequisites.

University. makeBidWithSignature(
bytes8 code,
uint amount,
bytes32 hash,
bytes memory signature
) public biddingIsOpen onlyStudent isCourse(code)

- Allows a student to make a bid for a course that they have not passed the prerequisite for, but they must provide a signed message from the course lecturer containing the University contract address, the public address of the student and the course code of the course being bid on. These conditions are checked with require statements.

Lecturers give their permission by signing a message using their Ethereum private key, that contains the following information: the Ethereum address of the student, and the course to which they are being granted permission.

- See **makeBidWithSignature** above.

Lectures may grant permission only for their own courses
- **makeBidWithSignature** specifically checks that the signer of the message is the lecturer of the course requested.

When submitting their bid for a course to which they do not have the prerequisites, students should include a copy of the signed message from the lecturer that grants them permission to enrol. The system should verify that the signature is correct before accepting the bid
- **makeBidWithSignature** verifies all this before accepting the bid.

Missing Requirements Implemented

A student should be able to remove a bid from a course if they change their mind. They can then use these tokens elsewhere.

University.removeBid(bytes8 code) public biddingIsOpen onlyStudent isCourse(code)
- Allows a student to remove a bid from a course

Running Costs to University

Function	**Cost (gas)**	**Cost ($AUD)**
Deploy Contracts	1329548	558.41
Add Admin	18012	7.56
Add Lecturer	8923	3.75
Initialize Contract	9701	4.07
Create Course	31057	13.04
Pass Student	8784	3.69
Start Round	13778	5.79
End Round	17569	7.38

These were calculated using gas-price=150 gwei, 1 ETH = $2800 AUD.

Deploying the contracts appears to be greatest cost to the university but the total cost of the other functions can grow quickly when they are called several times. The contracts support adding multiple admins in one transaction therefore although the gas cost will rise when adding more it should not cost as much as adding each admin separately. Such is the case with adding lecturers which must be added one by one by an admin. Fortunately the direct cost of enrolling students has been transferred to the student themselves. However as the number of students increase, the number of bids will increase even more and therefore much more bids will need to be processed in when a bidding round ends which will increase the gas cost rapidly. Furthermore, adding more courses is a sizable gas cost on its own, but again increasing the number of courses will increase the amount of processing required at the end of a bidding round. Finally, increasing the number of bidding rounds in a session multiplies this total even further as processing the end of a bidding round would happen several times. However subsequent bidding rounds should cost less to process due to students being accepted to courses and therefore being unable to bid as many times in the next round.

Reflection and Security Considerations

Overall, I don't believe the Ethereum platform is suitable for this use case. This is because Ethereum has high gas prices at the moment and the cost to the Uni will quickly multiply throughout the session as more courses, students and bids are added to the contract. Furthermore, the delay in making a transaction and then having it confirmed enough times does not lead to a good user experience in a bidding application. Finally with regard to security considerations, although blockchain technologies are vulnerable to a 51% attack which is when a malicious individual or group of individuals owns more than half of the total mining power on Ethereum, in recent times there has been a massive increase in miners on Ethereum so a malicious entity would need to own much more mining hardware to compete. Another aspect of security is the anonymity of users in the system. Whilst users are anonymous in the system via Ethereum addresses, there are points of vulnerability where their identity could be revealed. For example, communicating with the university to get involved or communicating with lecturers to receive exemption from prerequisites. Additional to this, the state of the system is visible to everyone and therefore a malicious actor could reverse engineer the identify of a student from the courses that they have enrolled in and bid on.