P2P and Blockchains, Final Term

Matteo D'Onofrio, 561901

06/06/2022

1 TRY Application Structure

The TRY application is composed by 3 smart contracts: *Collectibles.sol*, *newNFT.sol* and *Lottery.sol*. Let's see the main structure of each of them.

1.1 Collectibles.sol

The Collectibles contract is defined in order to create, sell and handle the collectibles. A collectible is defined by an integer ID and a short string that describes it. When the contract is deployed, 8 collectibles are created by the constructor (one for each prize class) and their description is set. About the state of the contract, the collectibles are stored in a mapping named `collectibles_list` and the cost of each of them is fixed by the variable `collectible_prize`. About the functions, a `buy_collectible()` external function is defined in order to allow the main contract (*Lottery.sol*) to buy the collectibles needed.

1.2 newNFT.sol

The newNFT contract is a derived contract developed by inheriting two base contracts named `nft-metadata.sol` and `ownable.sol` (source code at [1]). Specifically the contract nft-metadata.sol, which in turn inherits the official standard interface *ERC*721, provides a set of main functions like: `_mint()` that creates new NFTs objects, `_burn()` that destrory an NFT object, `_transfer()` which allow to transfer the ownership of a NFT; the contract ownable.sol has an `owner` address, and provides basic authorization control which simplifies the implementation of user permissions, like the `onlyOwner` modifier that check if an action is invoked by the owner, the event `OwnershipTransferred`. So about the state of the contract, in the newNFT contract we preserved some attributes from the super classes (the address of the owner named `owner` and the integer `token_id`), while others were dropped (the string `nftName` and the string `nftSymbol`) since being strings they require a lot of space and in our application they are not strictly needed. Then we have added two new attributes: a string describing the information of the collectible used to mint it,

[1] https://github.com/nibbstack/erc721/tree/master/src/contracts

named `description_collectible`, and an integer representing the class value assigned to the NFT, named `class`. About the functions we've defined our function `mint()` that invokes the super class function `_mint()` (this re-definition was done in order to store correctly all the attributes, both older and newer) and we've introduced an external function `give_to_winner()` that allows the transferring of the ownership of an already minted NFT (done by invoking the `_transfer()` method of the super class)

1.3 Lottery.sol

The main contract that handles the whole lottery is *Lottery.sol*. It provides the main functionalities of the application: the buying of the collectibles, the starting and closure of the rounds, the purchase of tickets, the extraction of the numbers and the prizes' assignment. About the state of the contract, the most important data structures are the following:

- the struct `Ticket` is composed of two attributes: the owner of the ticket and the set of 6 numbers chosen by him. All the tickets bought in each round are stored in a dynamic array `tickets_sold`.
- the mapping `collectibles_bought` storing all the collectibles bought to the Collectibles contract
- the array `drawn_numbers` used to store, in each round, the extracted numbers. In addition, a mapping `drawn_numbers_mapping` was defined in order to handle the lottery in a more efficient way (since allows to compare the `drawn_numbers` with the numbers provided by the users without executing loops).
- the array `NFT_minted` containing all the NFTs minted.
- the mapping `reward_list` which maps the address of the winner users with their respective class value prize.
- the immutable integer M defining the number of transactions that set the length of a round.

2 Contracts Security

In order to analyze the security of the whole application, we'll discuss two topics: firstly the functions' visibility and modifiers, then the randomness of the extracted values.

Before going ahead, we notice that the initial idea was to use the $SafeMath$ library in order to prevent any underflow or overflow problem, but looking to the Solidity's documentation it came out that "Arithmetic operations revert on underflow and overflow. You can use *unchecked* {...} to use the previous wrapping behaviour. Checks for overflow are very common, so we made them

the default to increase readability of code, even if it comes at a slight increase of gas costs." (link at). This update was inserted from the compiler version 0.8.0 on, for this reason in this application we adopt any compiler version greater or equal than 0.8.0.

2.1 Functions' visibility and modifiers

2.1.1 Lottery contract

In the Lottery contract, all the variables of the state are *private*, so they can be modified only by the functions defined inside the Lottery contract. All the functions that are declared with the function visibility *external*, adopt a function modifier `onlyOperator` that checks if the address that invoked it is authorized or not. This is true for all the external function except the `buy_ticket()` function, since it can be invoked by any user. In addition, since the whole application can be divided into 4 main phases: initial buying collectibles phase, user buying tickets phase, extraction phase and prizegiving phase; each *external* function also adopts some function modifiers that check if the request action can be done at that moment or not. These function modifiers are: `balance_receiver_set`, `buyPhaseActive`, `extractionPhaseActive`, `prizesComputedAlready`, `rewardPhaseActive`, `roundIsFinished` and `lottery_closed`. So overall all these modifiers enforce the flow of the application to the correct order.

Another security control is done on the function `give_prizes()` which assigns the NFTs to the respective winner. This is done because the operator could invoke this function multiple times so he could assign more NFTs than necessary. So a modifier `prizesNotGivenYet` is introduced in order to guarantee that this function can be invoked just once per round.

The same reasoning is done for the function `close_lottery()`. If this function is invoked during an active round (so during a phase in which users can buy tickets), each user receive back the amount spent. In order to prevent a re-entrancy attack (in which a fallback function is implemented in order to call again the function `close_lottery()`), is defined the modifier `lotteryOpen`, that allow to call this function only once.

These precautions implemented through the modifiers, together with the *private* visibility of the state variables, ensure that the code has a re-entrancy prevention.

At last we say that thanks to the function modifiers, after the invocation of the `close_lottery()` function, it's guaranteed that no more user can send ether to the smart contract through the `buy_ticket()` function.

2.1.2 Collectibles contract

In the Collectibles contract, the only function that modifies the state is `buy_collectible()`, it's *external* and checks that each collectible can be sold only one time, so only to a single address.

2.1.3 newNFT contract

In the newNFt contract, the `mint()` and the `give_to_winner()` functions can be invoked only by the lottery operator. In addition, since initially some NFTs are minted by the operator, in order to ensure that only him is able to change the ownership of each NFT, the `give_to_winner()` function use the `onlyOperator` modifier that checks that only the operator can invoke it.

2.2 Drawn number randomness

The extraction of the 6 random numbers is computed in the `draw_numbers()` function (which in turn calls the `draw_numbers_compute()` function). In order to compute each of these numbers, in each round is chosen a *seed*, whose value is the $block.difficulty$. Then a loop of 6 iterations starts (one iteration for each random value). Inside the loop a variable `extracted_value` is initialized to 0, then in each iteration its value is given by the random number generated at the iteration before. Overall, in each iteration the random number is computed as follows:

- $bhash = (keccak256(abi.encodePacked(block.number, \texttt{extracted\_value}, seed)))$
- $rand = keccak256(abi.encodePacked(bhash))$
- `extracted_value` $= rand \mod K$, where K is a given constant.

These sequence of computations has a main property, the determinism, which allow to reproduce the same sequence all around the blockchain, if the same input values are used. The crucial input values are *block.number* and $block.difficulty$, which are public and accessible in any time by any node in the blockchain.

The second property is the difficulty of prediction, since a malicious attacker should know in advance both the *block.number* and the $block.difficulty$ values before that the extraction starts. Despite this an attacker could perform an attack in the following way: starting from the moment in which a round start, he could take the initial *block.number* and the $block.difficulty$, then he could try to increase each of these values of a single unit, and for each combination generate a sequence of numbers and buy a ticket with that sequence. This attack should be done after the round start but before the extraction start and would require a lot of computational power; the drawback is the cost of the tickets since he should buy a lot of tickets without any guarantee of winning.

In addition the `extracted_value` has been added in order to decrease the probability to get two equal values consecutively. However the production of duplicates values can happens, so in the contract is implemented a check to the random values such that if in a single extraction exist duplicates, **the extraction will not be considered valid and the operator must start another one.**

It has been decided to not exploit the *block.timestamp* value, since it could be changed by the miners.

3 Event log

Here we show what are the *events* generated by the application during its execution.

- `start_new_round(bool)` used to communicate the beginning of a new round.
- `start_extraction_phase(bool)` used to communicate the beginning of the extraction phase in which the random numbers will be generated.
- `start_reward_phase(bool)` used to communicate the beginning of the prizes assignment.
- `ticket_bought(uint8 [6])` used to show the values submitted by a user.
- `values_drawn(uint8 [6])` used to show the random numbers generated.
- `prize_assigned(address, uint8 [6], uint8 [6], uint8)` used to show for each winner user: the address of the user, the values provided by the user, the drawn numbers and the class of the NFT that will be assigned.

4 Gas Estimation

In order to estimate a cost, we'll refer to the costs shown in the slide named "13-04-22-EthereumAccountGas". In the analysis we could distinguish two main kind of operations: the ones done in the memory (with low fee price, up to $\approx$ 10) and the ones done in the storage (with high fee price, up to $\approx$ 32000). In order to make a simpler but significant analysis, we'll concentrate on the ops done in the storage.

Let's estimate the execution cost of the function `close_lottery()`. We assume that the function is invoked after the first round has started and two tickets were already bought. So in the function only two tickets need to be refunded.

The body of this function performs the main following operations:

- firstly checks for the modifier correctness: so the `onlyOperator` modifier test the condition by comparing the variable `msg.sender` (contained in the calldata) vs the variable `operator` (contained in the storage). Then the `lotteryOpen` modifier test the value of the boolean variable `lottery_open` (contained in the storage). Overall the cost of two storage reads (for the `operator` and `lotteryOpen` variables) is $\approx 2 * 200 = 400$.
- then sets all the 5 boolean state variable to false, but since 3 of 5 were already false, only 2 of them are modified, so incurring in a higher and significative cost. Overall the cost of two storage writes is $\approx 2 * 20000 = 40000$.

- finally we've the loop, where we recall that is executed two times accordingly to the premises. First is read the variable `tickets_sold.length` from the storage, then in each iteration is read the variable `tickets_sold[i].owner` and is executed a `transfer`. Here we don't take into account the cost of the read of the variable `price_ticket`, because it's declared immutable so the compiler replaces the value with the variable name at compile time. Overall we've to sum the cost of a first storage variable in the guard, the cost of 2 storage variables in the iteration and the cost of 2 transfer, so we get $\approx 3 * 200 + 2 * 9000$ [2] $= 18600$.

So overall the cost for this function is ≈ 59000. In addition to this we've to consider the cost given by all the ops done in memory (like ADD, SUB, MUL, DIV, PUSH, POP, MLOAD, MSTORE etc...) and the gas fee paid to the miner.

5 List of operations

For the demo of the application, run the following commands:

- Open the Remix IDE and import the provided backup named `RemixRemix-Backup_P2P_Code_D'Onofrio 561901`. Then click "import" on the backup named `TRY_P2P`.
- select a compiler version $\geq 0.8.0$.
- select the box *enable optimization*.
- compile the contracts in the following order: newNFT.sol, Collectibles.sol, Lottery.sol.
- select any account, it will be used as *LotteryOperator* (since it will be used also later, is recommended for convenience to pick the first one). Then select, in the *Contract* drop down menù, the field $Lottery-contracts/Lottery.sol$ and click *deploy*.
- using the operator account, repeat 8 times the following procedure for buying all the 8 collectibles: put 1 ether in the $Value$ field; insert an integer $\in [1, , 8]$ in the input field of the `buy_collectibles` function; invoke the function. **NB: this procedure must be done 8 times, in each time a different integer input must be provided, in order to acquire all the 8 different collectibles, each identified by an integer id** $\in [1, , 8]$**.**

[2] This value was taken from the Solidity documentation page containing a link named "Ethereum Gas Prices", detailing that a $GCALLVALUETRANSFER$ costs 9000. Link at: https://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html

- select any other account that will be used as `balance_receiver` account, that is the account that collect all the contract balance at the end of the round. Using the operator account, put the address of the chosen account in the input value and invoke the `set_balance_receiver` function.
- using the operator account, invoke the `start_new_round` function.
- choose any other account (or more than one) and repeat iteratively the procedure for buying a collectible: put 1 ether in the $Value$ field; insert an array of 6 integers in the input field of the `buy_ticket` function; invoke the function. Repeat this procedure until the end of the round will be notified in output with the message "`start_extraction_phase`". Anyway if after this you try to buy another ticket an error message will be displayed.
- using the operator account, invoke the `draw_numbers` function.
- using the operator account, invoke the `compute_prizes` function.
- using the operator account, invoke the `give_prizes` function.
- in order to start another round, using the operator account, invoke the `start_new_round` function.

Any different behaviour could be tested by providing wrong values, invoking a function when it's not possible or using a wrong account to invoke functions.

6 Modifications

The actual application is deployed on Remix IDE. By the rules written in the Solidity language and in the Remix IDE, all the function that work in memory (allocating temporary data) should clean and remove the data just after the function returns. By looking at the behaviour of this IDE, it seems that the temporary variables are not removed instantly, but after a variable time. This leads to a huge memory occupation that still even after a function execution terminates. When this memory occupation become too high, the IDE usually crash and need to be restarted. In order to enable a flowing test of the application, some changes were made:

- Initially, before the first round starts, only one NFT is minted (the NFT with class 1). The others are minted when needed.
- When an NFT is minted, the rank is assigned in a deterministic way, in order to avoid a long loop execution until the right rank come out.
- The use of the strings data type has been reduced at the minimum, in order to not occupy a lot of memory but to guarantee that all the objects (Collectibles and NFTs) are uniquely identified.