**Zombie Battle System**

Zombie Battle System is the fourth course on cryptozombies.io. This course unveils two important concepts, one is the mechanism by which one can be made to pay a certain amount of ether to call a function (ether that can later be withdrawn, usually by the contract owner), another is a security concern raised by the difficulty of using random numbers in blockchain.

Payable is the modifier used to indicate a method can receive payment (other than gas) from the caller for any conceivable reason. The course gives as example a fee of 0.001 ether to attack a zombie whenever you fancy (unless under cooldown, of course). An important consideration is that you usually do not want fixed payments, you will want a mechanism to adjust the amount of ether to pay, since crypto-currencies are very volatile. The example implemented in the course simply allows the owner to set the amount of ether attacking a zombie will cost. This might be an issue if the owner decides to abandon their contract, but doesn’t disable it, in the future it might be prohibitively expensive to attack a zombie with the set price.

Of course, ether paid by the user will have to reach someone’s address, and be able to be used in future transactions eventually. Any payment made in this way will store the specified amount in the contract’s address, which can not be directly accessed other than by a function on the contract. The most obvious use case the course shows is that the owner is the only person that can withdraw the ether on the contract’s address to their own, which presents an obvious way one can make profit from this application. However, you could set up other payment schemes, perhaps the owner can modify a set of users that can withdraw certain amounts from the contract, or anything more complex. However, we can all agree the worst possible outcome is that the money will never be withdrawn and will remain on the contract’s address, which should never be allowed.

The other interesting issue addressed in the course is the dishonest node attack. Suppose you have a coin, and based on the outcome of its flip you either win or lose a billion ether. If you simply generate random numbers based on the unix timestamp at the time of the transaction, you allow a node on the blockchain to run the transaction, check whether it gives the desired result, if not then it can choose not to publish the result. If it runs again, then it can choose to publish the result, granted it will achieve proof of work (PoW) before all the other nodes. For such small examples (the zombie application) it is unfeasibly expensive to set up an army of nodes that process the transaction before the rest of the network, but for that one billion ether it might be worth it. In practice it is explained that we can overlook this apparent vulnerability as long as the stakes are not too high, but it highlights how many unexpected security concerns have to be addressed by smart contracts.

There is however a solution to the dishonest node attack. Suppose we have a gambling application that has multiple users involved in a round of a certain game. First the casino sets aside an award for the current round. Then each user will, locally, generate their own secret number N and send to the contract the hash of the concatenation between the number N and their address. Along with this hash each user will also deposit an amount of ether at least as large as the number N chosen by them. After all the submission are made, the reveal part begins: each user will be asked to submit their number N again to the contract. If it can be verified that the number N sent by the used matches the first value sent, then the submission is considered valid. If the user fails to do so, their deposit will be forfeit. After all numbers are submitted, the chosen Ns from all of the users will be used to generate the deciding random number. This solution is, of course, not perfect since the last user has a choice whether to reveal their N or not. They will be able to see the numbers chosen by each user and if they would lose they can just not play and get their deposit back. Another improvement is to force a trusted node (or the house) to submit the first and last N, but then that centralizes the system since they can shut down and then the game can no longer be played.

In conclusion, we learned how we can ask users to pay ether to call functions on our smart contract and what we can do with the ether spent by them, as well as security concerns raised by random numbers. All of this contributes to the complexity that blockchain developers have to deal with while developing their applications, and to make triply sure everything is air-tight before they can deploy.