COMP-4541 – HomeWork 4

Farshid Adavi - 4021305965007

---

security vulnerabilities and implementation bugs

Source of Randomness

Security vulnerabilities in random_uint16.
Blockhash, block.number and gasprice are not reliable sources for randomness.
Now let's see how it leads to an attack:
Suppose I am that miner who is going to put the last block in the blockchain.
I have 12 seconds to make many random numbers by changing gasprice.
Now I create some transactions with these gasprices and call the createNewDoggy function in them.
And I put these transactions in this created block and add it to the blockchain and become the owner of several expensive doggys.
Solution:
A new uint16 random function using Shamir's Secret Sharing idea where volunteers participate in generating random numbers.
Unfortunately the random generator implementation proved to be intricate, resulting in excessively large code and numerous compilation errors. Additionally, due to time constraints, there was insufficient opportunity to adequately comment and debug the code. I aim to rectify these issues in the resubmitted assignment without any violations.

Infinit gas

Poor implementation of createNewDoggy.
This implementation can also lead to bugs. In this 16-bit example, the number of doggies cannot be large enough to increase the gas consumption of the for loop from 30 million, but it is better to correct it.
Actually, instead of looping on the doggies, it would have been better to loop on the last thousand blocks. Of course, this was not necessary. Because it can be proved that:
Checking the doggies with the highest creation fee, generated by calling createNewDoggy in the most recent block that produced at least one doggy, is sufficient.
Suppose we are in Block B_n and the doggys with highest creationFee in the most recent block named X created in Block B_i So that $n - 1000 < i$.
Now, $\forall$ doggy Y with highest creationFee in block $B_j, j \neq i$ such that $n - 1000 < j$, we know that $j < i < n \rightarrow i - 1000 < j$.
Now, if creationFee Y is more than creationFee X, it is a contradiction. So, always the highest creationFee in the past 1000 blocks, is the highest creationFee in the most recent block.

Also, if the createNewDoggy function did not create any new doggy in the past 1000 blocks, creationFee returns to the base value. Solution:
Remove mapping birthBlock snd paidCreationFee.
We retain only the last block number in which the doggy was created, as well as the amount of the highest paid creation fee paid in that block.

Identical doggies

Poor implementation of createNewDoggy.
Creating two identical doggies can lead to numerous issues within our mappings. For instance, problems may arise during selling and...
Solution:
1. Checking for identical twins allows for preventing production. 2. Doggies can be assigned unique names, ensuring all items are saved under distinct identifiers.

Fishing by 'tx.origin'

Security vulnerabilities of tx.origin in createNewDoggy() and sellDoggy().
Now let's see how it leads to an attack:
I've got a contract that lacks a receive() function. To collect owed funds, I'll provide the debtor with this contract's address. Then, in the fallback() function, I execute createNewDoggy() once to create a doggy. Subsequently, I list the created doggy for sale at the minimum price using sellDoggy(), and buy it back with buyDoggy(). If the gas costs incurred in this process exceed the buying fee, I might pass those costs on to the money sender.
However, the presence of tx.origin in sellDoggy() poses a more severe risk, potentially enabling me to list the sender's valuable doggy for sale at a minimum price and repurchase it myself.
Solution:
replace 'txȯrigin' with 'msgṡender'.

Creation Fee

A setup where only the last 1,000 blocks affect creation cost increases raises several concerns, especially given that all costs go to the developer:
This setup encourages the developer to avoid periods of inactivity to maintain the rising Creation Fee, potentially increasing their profits.
Additionally, when all Creation Fee are passed on to the developer, they may consistently produce valuable doggies and amass a significant holding.
Solution:
Given these issues, if the purpose of the Creation Fee increase is to reduce high demand pressures, the first concern can be mitigated by keeping the developer's income constant. In addition, setting the revenue below the Creation

Fee can discourage the developer from continuing to create the doggies.

Wrong ownership for puppy

Poor implementation:
A doggy that is in the process of breeding should not be sold.
Or a doggy that is in the process of breeding can be registered for sale.
Because if the owner of the doggy changes during breeding, the new puppy may reach the buyer.
Solution:
Define a lock to prevent wrong ownership.
And also remove the genetic code [00000000000000] from the domain and so that we can check that the person who paid the breeding fee does not sell the doggy (also useful for other cases).

Deniledenial of service

Security vulnerabilities of breedDoggy():
Suppose the first person paid 1 Ethereum and executed breedDoggy(). Now the second person does not execute breedDoggy() and intends to extort.
Solution:
Implementing a mechanism where the first person can cancel this action or change the mate.

Failure to pay sellingFee

Poor implementation, sellDoggy() is not payable.
This money can be deducted from the sale amount. Of course, the sale amount may not be enough. It is also possible to make sellDoggy() payable, but in this case it is not possible to cancel the sale or change the price.
Solution:
Implementing a mechanism where person can cancel this action or change the price.

Terrible problems in receive_Money()

Poor implementation of receive Money() lead to many problems that are almost the same.
The following two scenarios for example:
1. Losing money from selling doggy:
Before withdrawal, the doggy is resold and the address of the previous seller is lost.
2. Security vulnerabilities:
Stealing all the balance of the contract

I can sell the doggy to myself or to another address that belongs to me.
Then put it up for sale for an amount equal to the entire balance of the contract.
And then call receive Money().
Solution:
Implementation of a domestic bank.
Increasing the seller's balance when selling successfully.
Ability to withdraw from the account.

About the achievements of this domestic bank

This bank has helped to recover the amount paid as a fee for an action in case of cancellation of any action.
Fees for actions are paid to the contract account at the moment of request for an action, but they will not be deposited into the developer's account until the moment the actions are finalized, and if any action is canceled, the fee amount can be withdrawn.
There is a non-withdrawal account next to the main account for managing fees.
The main account contains the money obtained from the sale of doggy or the returned amounts from the cancellation of the requested actions.
But the non-refundable account includes the fees that we have deposited for pending actions, and if the action is finalized, they will be deducted from the account and transferred to the developer's account.
Also, the developer has a withdrawal account where the deposit fees go to that account, and like other players, he can make withdrawals with the receive money function. And I no longer needed the reclaimFees function and deleted it.

Bug in random_offspring()

According to the question, each puppy should randomly inherit half of her genetics from each doggy.
But in this function, it inherits equal the number of 1 in the random number from the first doggy and the rest from the second doggy.
Solution:
We need a random number with the number of 1's and 0's that come from a uniform distribution.
Uniformity of this distribution makes it difficult.
In fact, if we check, we will see that it is not possible to simply start picking genes from the first and second doggies based on 1s and 0s in a 16-bit random number, and wherever 8 genes are selected from one doggy, the remaining genes are selected from another. Yes, with this, masks will no longer be part of the uniform distribution.
Solution1:
For each gene combination, select eight random numbers and calculate the first one modulo 16, the second one modulo 15, and so on, with the last one calculated modulo 9. Label these eight answers as r1 to r8. Next, starting from the first doggy, copy the gene at position r1 and place it in the corresponding position within the genetic code of the puppy. Then, utilizing the remaining 15 genes from the first doggy and the puppy, repeat this process for r2

to r8. Finally, fill the remaining positions in the genetic code of the puppy with genes from the same positions in the genetic code of the second doggy.
This solution will increase the cost for us in the long run, but it is better at the beginning than the next one.
Solution2:
Combination 8 of 16 equals 12,870, is the number of all available masks.
It stores 16x of 16-bit masks in each 256-bit word of storage.
So we need 805 word.
each word cost 5000 gas.
Therefore, the hard coding of all filters requires about 4 million gas, which is less than 400 dolars or about 0.1 Ethereum at the current price.
In this case, one of the filters can uniform randomly be selected, only by generating one random number and calculating its remainder to 12,870.
which reduces our costs in the long run.
Even if the genetics are so long that hardcoding the filters requires more than 30 million gasses, this is still possible.
In this way, we first put the masks in separate contracts on the block chain.
Then, in the main contract, after generated one random number, we calculate which word block number and which word shulde be selected. Actually, blocks on block chain are available to us like as different directory.