COMP4901W Homework 6

hson@connect.ust.hk

Hangyul Son

20537267

Problem 1:

tx.origin to msg.sender. Use of tx.origin opens the vulnerability to pishing attacks.

Solution: Change all tx.origin to msg.sender

Problem 2:

Expected Randomness. Potential doggy owners can create a dummy contract to check value of the random number before executing the createNewDoggy(). Also miners can exploit randomness as the function make use of block.number and blockhash. Therefore the randomness is now dependent to the miner.

```
uint random = uint(blockhash(block.number))^tx.gasprice;
```

Solution: Randomness cannot truly implemented within the blockchain. Therefore, use chainlink oracle API to receive random number off chain. Or make use of lottery input scheme for randomness. However, implementation of either method would cause a huge change in the code. As a result, the implementation has not been taken place in the modified version of the contract.

Problem 3:

Mapping variable paidCreationFee can create overflow if multiplied with 101 and be interpreted as small number. The attacker can utilize this vulnerability and send small value to create a doggy. Also, the similar overflow vulnerability can occur for msg.value * 100.

```
require(paidCreationFee[doggies[i]] * 101 <= msg.value * 100);
```

Solution: Check if there is an overflow. If the multiplied value is less than original value, overflow has occurred

Add line of code below,

```
require(msg.value * 100 > msg.value);
require(paidCreationFee[doggies[i]] * 101 > paidCreationFee[doggies[i]]);
```

[EXTRA POTENTIAL OVERFLOW]

In addition the original code for the for loop,

```
for(uint i=0;i<doggies.length;++i)
```

if uint is assigned 0, uint type might be assigned as uint8 which would then not be able to hold valid value for ‘i’ if overflows occur when doggies.length number becomes very large.

Therefore, to prevent overflows with variables within for loops, make for loop declared variables as uint256.

```
uint256 i=doggies.length-1;
```

Problem 4:

Vulnerability of having out of gas possibility within the for loop if the for loop starts from the first doggy.

```
for(uint i=0;i<doggies.length;++i)
```

Solution: Mitigate the possibility by starting the for loop from the latest added doggy. Stop the for loop as soon as the doggy has been created before block.number – 1000.

```
for(uint256 i=doggies.length-1; i>=0 && birthBlock[doggies[i]] >= block.number-1000;i--)
```

Problem 5:

breedDoggy() function fails to check if breed has been completed. This leads to the vulnerability in which the user repetitively can breed the doggy with only 1 ether instead of 2. With 1 ether create 2 doggies.

Solution: Initialize the currentMate mapping for my_doggy and other_doggy as 0.

```
currentMate[other_doggy] = 0;  currentMate[my_doggy] = 0;
```

Problem 6:

receiveMoney() function vulnerable to reentrancy attack.

```
recipient.call{value: price[my_former_doggy]}(""); //pay the sale value to the previous owner
```

Solution: Make use of transfer() to send ether in order to prevent reentrancy attack which is hardcoded. Also before the execution of transfer() set the price[my_former_doggy] to 0.

```
price[my_former_doggy] = 0;

recipient.transfer(amount);
```

Problem 7:

If developer execute reclaimFees() before seller executes receiveMoney(), seller cannot receive the money as the reclaimFees() function returns the entire balance of the contract instance.

```
developer.transfer(address(this).balance);
```

Solution: Create a variable 'developerBalance' which keeps track of the amount in which the developer must receive overall. Also prevent reentrancy attack by the developer himself.

developerBalance variable is increased for buying fee, breed fee, selling fee, and buying fee.

```
uint256 developerBalance = 0 ether;


function reclaimFees() public
  {
    //we do not need access control since the fees will be paid to the developer anyway (no matter who calls this function)
    uint256 amount = developerBalance;
    developerBalance = 0;
    developer.transfer(amount);
  }
```

Problem 8:

sellingFee is not implemented when the doggy owner receive money after he successfully sells his doggy.

Solution: Implement the sellingFee before doggy owner receive money from the sales.

```
uint amount = price[my_former_doggy] - sellingFee;
```

Problem 9:

The price of doggy can potentially be less than the sellingFee. This means that the selling fee will not be collected.

Solution: The asking_price in which can be registered must be greater than the sellingFee.

```
require(asking_price > sellingFee);
```