# COMP4901W Homework 6

KIM, Jaehyeok (20550178)

The Hong Kong University of Science and Technology

## Exercise 1

1. For `createNewDoggy` function, it is inefficient to check from index 0 to the end as the number of doggies is not constrained and this may end up out-of-gas. Moreover, if `uint i` is initialized to 0, it could be initialized to `uint8` type. Therefore, I modified it to traverse from the end to the beginning. If `(birthBlock[doggies[i]] >= block.number - 1000)` is false, I changed it to break the loop since doggies is kept in a chronological order. This version is safer from the denial-of-service attack than the previous version.

2. Even within the `for` loop in `createNewDoggy` function, multiplying 101 and 100 respectively to `paidCreationFee[doggies[i]]` and `msg.value` can result in overflow. Therefore, I added two condition checks as followed before comparing two multiplied values.
   ```
   require(paidCreationFee[doggies[i]] * 101 >= paidCreationFee[doggies[i]]);
   require(msg.value * 100 >= msg.value);
   ```

3. For `require(owner[my_doggy] == msg.sender);` in `sellDoggy` function, the validation should not be done by comparing `tx.origin` that returns original address that started the transaction. A potential attacker can create another contract to phish people to invoke his contract and sell their doggy at the price that attacker wants. The victims might sell their doggies at the price that they didn't want. This is resolved my replacing it with `msg.sender`.

4. In the `constructor()`, the address of the `developer` should be set by `tx.origin` instead of `msg.sender`. This is to set the address who kicked off the creation transaction to be the developer of the contract instance. There could be a possible attacker being in the middle to become the actual developer so that the reclaimed fees will be paid to the attacker.

5. `recipient.call{value: price[my_former_doggy]}("")` from `receiveMoney` function is vulnerable to reentrancy attack since it triggers the fallback function of the address. Therefore, it is replaced by transfer that is hardcoded to avoid reentrancy attacks. It also has a gas limit. To prevent multiple times of money receival, it first saves the amount of money, set `price[my_former_doggy]` to zero, and then transfer the saved amount of money.

6. When seller sells his doggy, the current contract does not charge the selling fee. Therefore, in `receiveMoney` function, `sellingFee` is subtracted from the amount of money to be transferred.

7. Following previous correction, `asking_price` must be greater than or equal to the `sellingFee`. Therefore, in the `sellDoggy` function, I set another condition checking with the code below before setting the asking price.
   ```
   require(asking_price >= sellingFee);
   ```

8. When buyer buys a doggy for sale, the amount of money is kept in the contract's balance until seller invokes `receiveMoney`. If the developer invokes `reclaimFees` during that time, the seller will not be able to get the money as it is hijacked by the developer with the following code: `developer.transfer(address(this).balance);`. To fix this problem, I created another variable called `uint256 devBalance = 0 ether;` to keep track of the developer's balance. In `createNewDoggy` and `breedDoggy` functions, `msg.value` amounts (`creationFee`, `breedingFee`) are added to `devBalance`. In `receiveMoney` and `buyDoggy` functions, `sellingFee` and `buyingFee` are respectively added to `devBalance`. In `reclaimFees` function, `devBalance` is saved to a temporary variable and is set to 0. Then, the developer get transferred `devBalance` amount of money.

9. For `breedDoggy` function, once two doggy owners bred once, they can breed again by only one of them invoke the function again since the `currentMate` is not reset. Therefore, when breeding is finished both `currentMate[my_doggy]` and `currentMate[other_doggy]` are set to 0.

10. The `random_uint16` function is not actually random. The random number is generated by using `block.number` and `tx.gasprice` that can be determined and priorly calculated by the miner. To securely generate a random number, a seed needs to be sent off-chain resources such as oracle. As implementing this feature would be too confusing to be graded, I left this security vulnerability as it is. I also left comment in the codes.