LESSON SC 11 - Tricks

University of West Attica
Department of Electrical and Electronics Engineering
Ioannis Christidis
Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

In this lesson we will not learn anything new, but we will discuss mindset.

How should we think when developing a SC?

The most critical thing to consider is that if we make a mistake in the SC code and the SC is deployed, we cannot do anything to fix it! So, we need to be extra careful when writing SCs. I bet that you already have one or more situations in mind where our SC's logic can break.

We cannot cover every aspect of that in this course, but we can do a few tricks to fix some of these points.

Here is an example. What will happen if a microcontroller address is added to multiple cars? It could result in the occupation of more than one car, which is not what we want. This can happen by accident or on purpose.

Since the owner is responsible of adding the addresses of the microcontrollers, this is an actual situation that could happen.

So, to fix this we will make some adjustments to the code.

Fixing the same address situation

To fix this we will create another `mapping` that tracks if an `address` has been registered in a car as a microcontroller.

```
    mapping (address _mc => bool _used) internal mcIsUsed;
```

Now in the `registerCar` function we can check if the `_mc address` has been registered:

- If it is, we throw a `custom error`.
- If not, the registration continues, and we also update our new `mapping`.

```
    function registerCar(address _mc, uint256 _price) external {
        if (mcIsUsed[_mc]) {
            revert ICRRegistry__McIsAlreadyUsed(_mc);
        }
        // rest of the checks
        mcIsUsed[_mc] = true; // mark mc as used
        Car memory car = Car(_mc, _price, Status.UNAVAILABLE, 0);
        uint256 currentCarId = nextCarId;
        cars[currentCarId] = car;
        nextCarId++;
        emit CarRegistered(currentCarId, _mc);
    }
```

We should also think of ways to check if the addresses are indeed microcontroller addresses and not random addresses entered by the owner, but we will do this later.

Freezing SC

Next let's assume that we have a bug on our code, and we find it, but we already deployed the SC. What can we do? We can create a way to freeze the functionality of the SC using a `bool` state variable.

```
    bool internal paused;
```

Now we can check before calling the function if the `paused` `bool` is true and if it is, revert, else, proceed. We can use it in the `rentCar` for example.

```
    function changePause() public {
        if (msg.sender != owner) {
            revert ICR__IsNotOwner(msg.sender);
        }
        if(paused) {
            paused = false;
        } else {
            paused = true;
        }
        emit PauseChanged(paused);
    }

    modifier isPaused () {
        if (paused) {
            revert ICR__CarRentingIsPaused();
        }
        _;
    }

    function rentCar(uint256 _carId) external payable carIsValid(_carId) isPaused() {/* functionality */} // rest of the functions
```

Other considerations

There are a few other things we could check to make the SC safer. Here are some examples:

1) We can check if `microcontroller addresses` have enough ETH to make a transaction and if they don't, the cars they are registered to cannot be rented.
2) We could alter the SC so that the owner controls what is the next status update a microcontroller applies to the car it is registered (`AVAILABLE` or `UNAVAILABLE`). This would be very use in case the car is stolen for example.

Best practices when writing SCs

It is generally a good practice to use a specific naming strategy for our variables so that the code is more readable. A developer reviewing the code can quickly understand how a variable is used, even without diving into its declaration.

`"i_"` for `immutable variables`: `immutable variables` are set only once during SC deployment and cannot be changed thereafter. Using a specific prefix ensures that developers can easily distinguish these variables and understand their behavior at a glance.

`"s_"` for `storage variables` : This differentiates `storage variables` from `local` or `memory` variables, which are temporary. This helps developers immediately recognize the scope and cost implications of interacting with this variable.

All-uppercase naming for `constants`: `constants` cannot change throughout the execution of the SC.

Best practices when writing SCs

We will also follow best practices and change the variables of our SC.

```
    address internal immutable i_owner;
    bool private s_paused;

    uint256 internal constant RENT_TIME = 1 days;
    uint256 private constant LOWER_PRICE_LIMIT = 0.1 ether; // 0.1 ether = 10^17 wei = 10^17
    uint256 private constant UPPER_PRICE_LIMIT = 10 ether;
    uint256 internal constant BALANCE_FOR_MC = 0.05 ether;

    mapping (uint256 _carId => Car _car) internal s_cars;
    uint256 internal s_nextCarId;

    mapping (address _mc => bool _used) private s_mcIsUsed;
```

Remember to change them everywhere in the SC.

Outro

Nice, we did some changes, and we discussed important things about the way we should think when we are developing SCs.

In the next lesson we are going to build the SC that will be used from the *Innovative IoT company (IIoTC)*.