# Updating Smart Contracts

Justus Perlwitz

2018-04-27

## Contents



As time goes on, smart contract requirements change. Often, a feature is added or a bug is found. But, once a smart contract has been deployed on the blockchain, it can't be upgraded. The entire value proposition of a blockchain is immutability. Yet, in the real world, contracts change all the time. For example, a labor contract will change when new labor laws are introduced. Similarly, there are many circumstances under which smart contracts change as well, such as

- smart contract features being added or removed,
- efficient implementations being found that lower transaction costs, and
- implementation bugs being fixed.

A contract itself cannot be replaced while still keeping the same contract address on the blockchain. Several methods exist that all use some kind of indirection that allow smart contracts to be replaced with newer versions and still use the same address.

## 1 1. Contract Relay

A contract relay forwards all incoming calls to a designated address using the `DELEGATECALL` opcode.

An example implementation can be seen in this StackExchange post.

The main idea is to define a fallback function that forwards all function calls.

```
contract Relay {
    address contract;

    // ...

    function update(address _newContract) {
        contract = _newContract;
    }

    //...

    function(){
        if(!contract.delegatecall(msg.data)) throw;
    }
}
```

`contract.delegatecall(msg.data)` will forward all transaction data, including `msg.sender`. For the receiver of the delegate call, the incoming call looks like it was placed by the user themselves.

The above mentioned post also links to a fuller implementation on GitHub. It addresses the problem of how to handle the contract's internal state when upgrading to a new version. Simply pointing the `contract` address variable to a new contract will mean that the old contract's data will not be copied over.

## 2 2. Contract Registry

The other approach to updating contracts is to break up contracts into smaller pieces that can be exchanged easily. For this, each part needs to implement a specific interface. If newer versions of the contract stick to the same interface, the caller does not have to worry about internal changes of the contract and can have static guarantees about the external behavior of this contract.

```
interface ContractInterface {
    // ... define your contract's interface here
}
```

Then, we create the contract that we will eventually upgrade.

```
contract Contract implements ContractInterface {
    // ... first implementation of the contract
}
```

The Registry contract will contain all the small contract parts that can be updated. Here, we allow the OpenZeppelin owner of the Registry to set the new version.

```
// ... import Ownable.sol

contract Registry is Ownable {
    ContractInterface public contract;


    // In this case, only the contract owner can upgrade the contract
    setContract(ContractInterface _contract) ownerOnly{
        contract = _contract;
    }
}
```

Then, in web3 we can add the contract to the registry.

```
// ... deploy the new contract
// ... find out the contract address and add it to the registry
await registry.methods.setContract(contractAddress).send();
// ... now we will get the contract address to use it
contractAddress = await registry.methods.contract();
```

Let's say that after a while you create a new contract and what the registry to point to it. First, the contract is defined using exactly the same interface as before.

```
contract NewContract implements ContractInterface {
    // ... new implementation
}
```

As soon as the contract is deployed, we — the owner of the Registry — can now point to the new contract by calling the `.setContract()` method with the new contract's address.

```
// ... deploy the new contract
// ... find out the contract address
registry.methods.setContract(newContractAddress).send();
```