Lab 4 – Interaction Between Solidity Contracts

COMP6452 Software Architecture for Blockchain Applications
Dilum Bandara

2023 Term 2

1 Learning Outcomes

In this lab, you will learn how to interact with multiple smart contracts, some of which may already be deployed. After completing the lab, you will be able to:

- develop a smart contract that calls functions in another smart contract
- understand transaction and execution context and its affect on the ledger state
- test multiple smart contracts by issuing transactions

2 Introduction

Many decentralised applications (DApps) require interaction among multiple smart contracts, including those deployed by third parties. For example, an escrow smart contract may lock in fungible tokens from an ERC-20 contract until payment conditions are satisfied. To swap a non-fungible token (NFT) in an ERC-721 contract against the payment of a fungible token in an ERC-20 contract, the atomic swap smart contract needs to connect to both token contracts. Therefore, the ability to call another smart contract's functions is essential in some use cases. It can also simplify programming and enhance smart contracts' upgradeability. However, calling another contract's functions could easily go wrong, resulting in deadlocked smart contracts, security vulnerabilities, or loss of assets. For example, in 2016, The DAO lost ETH 3.6 million due to a bug in its code that called other contracts.

This lab focuses on developing smart contracts that call functions in other contracts. We discuss several ways a *caller* smart contract can call/invoke a *callee* smart contract's functions depending on the level of information you have about the callee contract.

This lab has two parts. In Section 3, we will develop two simple contracts to illustrate how a contract can invoke another contract's functions and the resulting state changes. In Section 4, we extend the MyToken contract to enable a third-party contract to transfer ERC-20 tokens on behalf of an account.

3 Illustrative Example

Figure 1 illustrates the interaction between an external user account (EA) and two contracts named Proxy and Util. A *proxy* acts as an intermediary between a client requesting a service and the software providing that service. The user invokes three functions on the Proxy contract, which in turn calls the Util contract's `sum` function using three different methods. Transactions (aka message calls) are labelled with each transaction's `msg.sender`. The two tables illustrate each contract's storage layout. It lists the order in which variables are arranged within the contract's storage on the ledger. Storage locations on the Ethereum ledger are called *slots*. Each slot is uniquely identified/addressed using a 256-bit key and can contain up to 32 bytes. For example, the variable `lastSum` is stored in the first and third slots of the Util and Proxy contracts, respectively.

The following code segments show the Util and Proxy contracts, respectively. The Util contract has a function named `sum` that adds two integers. Before returning the results, it saves the total in `lastSum` and the transaction sender's address in `lastCaller`.

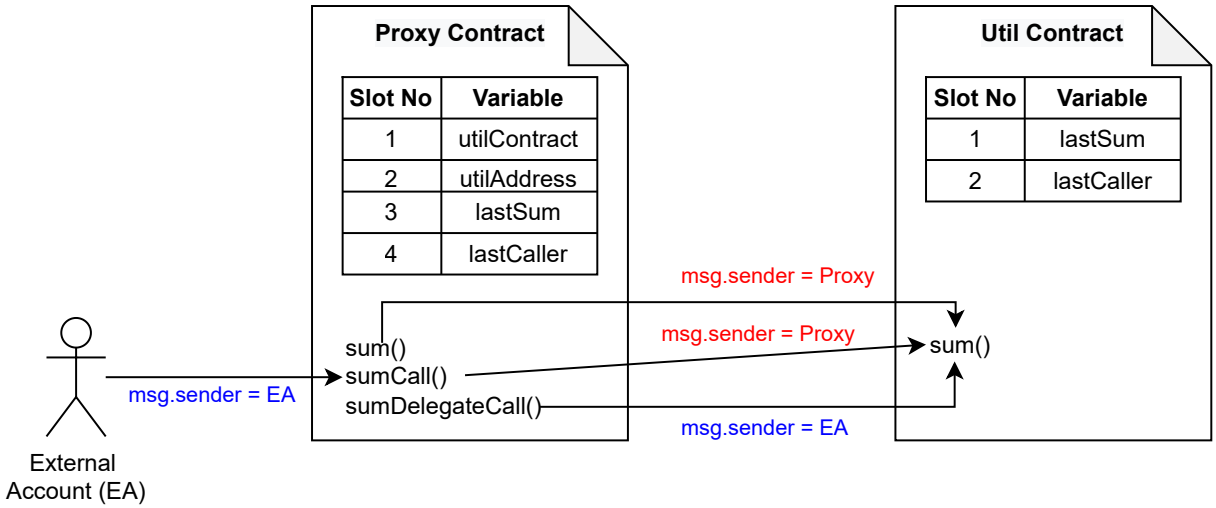

Figure 1: Interactions between an external user account and two contracts.

```solidity
/// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

/// @title Utility contract
/// @author Dilum Bandara, CSIRO's Data61

contract Util{
    int public lastSum; // Remember last result
    address public lastCaller; // Remember last address that called

    /**
     * @dev Get sum of 2 numbers while remembering the last caller & sum
     *
     * @param a First value (integer)
     * @param b Second value (integer)
     * @return Sum of 2 values (integer)
     */
    function sum(int a, int b) public returns (int){
        int total = a + b;
        lastSum = total;
        lastCaller = msg.sender;
        return total;
    }
}
```

```solidity
/// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.0;

import "./Util.sol";

/// @title Proxy contract
/// @author Dilum Bandara, CSIRO's Data61

contract Proxy{
    Util public utilContract; // Util contract
    address public utilAddress; // Address of Utility contract
    int public lastSum; // Remember last result
    address public lastCaller; // Remember last address that called

    /**
     * @dev Sets values for {addr} when the contract starts.
     *
     * @param addr Address of Utility contract (address)
     */
    constructor (address addr){
        utilAddress = addr;
        utilContract = Util(addr);
```

```
24      }
25
26      /**
27       * @dev Get sum of 2 numbers using Utility contract
28       *
29       * @param a First numbers (integer)
30       * @param b Second number (integer)
31       * @return Sum of 2 numbers (integer)
32       */
33      function sum(int a, int b) public returns (int){
34          return utilContract.sum(a,b);
35      }
36
37      /**
38       * @dev Get sum of 2 numbers using Utility contract using a call
39       *
40       * @param a First numbers (integer)
41       * @param b Second number (integer)
42       * @return Sum of 2 numbers (integer)
43       */
44      function sumCall(int a, int b) public returns (int){
45          (bool success, bytes memory result) =
46            utilAddress.call(abi.encodeWithSignature("sum(int256,int256)", a, b));
47          require(success, "Call failed");
48          return abi.decode(result, (int256));
49      }
50
51      /**
52       * @dev Get sum of 2 numbers using Utility contract using a delegatecall
53       *
54       * @param a First numbers (integer)
55       * @param b Second number (integer)
56       * @return Sum of 2 numbers (integer)
57       */
58      function sumDelegateCall(int a, int b) public returns (int){
59          (bool success, bytes memory result) =
60            utilAddress.delegatecall(abi.encodeWithSignature("sum(int256,int256)", a, b));
61          require(success, "Deligatecall failed");
62          return abi.decode(result, (int256));
63      }
64  }
```

The Poxy contract has a constructor and three functions. The constructor expects the Util contract address. To obtain the Util contract's address, it must be deployed before the Proxy contract. In line 22, the address is recorded in `utilAddress`. In line 23, we cast the address to a Util contract object. This is possible only when the callee contract's ABI (application binary interface) is known to the caller contract. Note that we import the Util contract's source code in line 5. A contract's *ABI* is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. Data are encoded according to their types. The encoding is not self-describing and requires a schema to decode. Similarly, functions and arguments are encoded in the ABI. You can find a contract's ABI in Remix IDE's `Solidity compiler` pane.

In lines 33-35, we call/invoke the Util contract's `sum` function, similar to how we call an object's method in object-oriented programming. This is possible only because we have access to the Util contract's ABI, and its address was casted in line 23 before setting the `utilContract` variable.

Lines 44-49 use Solidity's `call` function to invoke the Util contract's `sum` function. In line 46, the contract is identified from its address. The call function is useful when we do not have the callee contract's ABI. Instead, we indicate the function signature within the call function. While the `sum` function accepts two integers, we have to explicitly use `int256` data type instead of `int`, which is an alias for `int256`. There should be no spaces between the parameter data types. The function signature is then encoded using the `abi.encodeWithSignature` built-in function. The `call` function returns two values, a boolean value indicating the call's success or failure and any results returned by the called function. We check for success on line 47. If the call succeeded, the following line decodes the result and returns it to the user.

When the called function is read-only (aka view function), we should use `staticcall` function in place of the `call`. The EVM (Ethereum virtual machine) ensures that a `staticcall` cannot change the ledger state by accident.

Lines 58-63 use Solidity's `delegateCall` function to invoke the Util contract's `sum` function. The rest of the code fragment is the same as `sumCall` function. However, there are two notable differences between

the behaviour of `call` and `delegateCall` functions.

First, note the `msg.sender` values for `call` and `delegateCall` functions in Figure 1. While the `msg.sender` for `delegateCall` is the same as the external account's address EA, it changed to the Proxy contract's address in the `call` function. That is, while a `delegateCall` preserves the address of the external account that originated the transaction, the `call` replaces it with the caller contract's address. Because the `msg.sender` is part of the transaction context (another properly is `msg.value`), we can say that while the `call` changes the transaction context, `delegateCall` retains it.

Second, `call` and `delegateCall` functions have different execution contexts. *Execution context* is a concept in programming that describes the environment in which code is executed. It is the area within which an object acts. When `call` is used, the execution context is the Util contract. Hence, any ledger state changes will be within Util contract's state space. Conversely, `delegateCall` changes the execution context to the calling contract, i.e., Proxy contract. Hence, any ledger state changes will be within the Proxy contract's state space. Because we also have `lastSum` and `lastCaller` variables in the Proxy contract, those should get updated than the ones in the Util contract.

Let us execute these two smart contracts to see how transaction and execution context changes affect the ledger state.

Step 1. Go to Remix IDE. Create two contract files named `Util.sol` and `Proxy.sol`, and copy the above code.

Step 2. Compile and deploy the Util contract to `Remix VM (Shanghai)`. Note down the contract address.

Step 3. Compile and deploy the Proxy contract to `Remix VM (Shanghai)`. Make sure to enter the Util contract's address as the constructor parameter.

Step 4. Invoke the Proxy contract's `sum` function by entering two integers and clicking on the `transact` button. See the transaction output to check the transaction results, which should include the sum of the two numbers.

Click on `lastSum` and `lastCaller` buttons on both contracts to see which ones got updated. Only the state on the Util contract should get updated.

Step 5. Invoke the Proxy contract's `sumCall` function by entering two integers and clicking on the `transact` button. Check the transaction output to see the results.

Click on `lastSum` and `lastCaller` buttons on both contracts to see which ones got updated. Only the state on the Util contract should update. You may notice that `sum` and `sumCall` functions behave the same. That is because, underneath, invoking the `sum` function in line 34 works similar to the `call` function.

Step 6. Invoke the Proxy contract's `sumDelegateCall` function by entering two integers and clicking on the `transact` button. Check the transaction output to see the sum of two numbers.

Click on both contracts' `lastSum` and `lastCaller` buttons to see which ones got updated. You will see neither contract's state got updated, though we were expecting the Proxy contract's `lastSum` and `lastCaller` to be updated.

Try calling the `sum`, `sumCall`, and `sumDelegateCall` functions again. All transactions should fail. This is because we corrupted the storage slots in the Proxy contract. Click on `utilContract` and `utilAddress` buttons to note that those variables now have the external account's address and the sum of the two numbers, respectively. This is an example of how things might go wrong with a `delegateCall`.

EVM does not remember variable names or try to search them by symbolic names. Instead, it relies on storage slot numbers, as seen in Figure 1. While `lastSum` and `lastCaller` occupy slots 1 and 2 in the Util contract, they occupy slots 3 and 4 in the Proxy contract. Therefore, when we invoke the `delegateCall` it updates slots 1 and 2 in the Proxy contract instead of slots 3 and 4, corrupting the Util contract's address. This is why all subsequent execution of the Proxy contract's functions fails.

Step 7. Swap lines 11-12 and 13-14 in `Proxy.sol` to align the storage slots of the two contracts. Repeat Steps 2 to 6. You should see `lastSum` and `lastCaller` on the Proxy contract get correctly updated, and the contract does not get stuck.

4 MyToken with Third-Party Transfer

Now that we understand different ways of invoking functions on another contract, let us try an example that is likely to occur with the token transfer. This section uses the third-party token transfer feature in the ERC-20 standard. A token owner can designate a third-party address to transfer tokens on its behalf using the following function:

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

The first parameter specifies the nominated third party's address and the second parameter sets an allowance for the third party to limit the number of tokens it can transfer. Once a token allowance is approved, the third party can use the following function to spend those tokens:

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

The first parameter specifies the token owner's address, the second one is the token receiver's address, and the last one is the token amount.

Step 8. Make sure `MyToken.sol` contract from Lab 2 exists in Remix IDE. If not recreate it.

Step 9. Go through the OpenZeppelin documentation at https://docs.openzeppelin.com/contracts/2.x/api/token/erc20 to understand how the `approve`, `allowance`, and `transferFrom` functions are implemented.

Step 10. Create another contract and name it as `ThirdPartyTransfer.sol`. Add the following code to it. This contract invokes the `transferFrom` function on the ERC-20 contract (inherited by the MyToken contract) as a third party. For example, the ThirdPartyTransfer contract could be extended to support atomic swapping of two fungible/non-fungible tokens by calling `transferFrom` functions from respective token contracts.

```
1  /// SPDX-License-Identifier: UNLICENSED
2
3  pragma solidity ^0.8.0;
4
5  /// @title Third-party transfer contract
6  /// @author Dilum Bandara, CSIRO's Data61
7
8  contract ThirdPartyTransfer{
9      address public tokenAddress; // Address of MyToken contract
10
11     /**
12      * @dev Sets values for {addr} when contract starts.
13      *
14      * @param addr Address of MyToken contract (address)
15      */
16     constructor (address addr){
17         tokenAddress = addr;
18     }
19
20     /**
21      * @dev Get sum of two numbers using Utility contract using a call
22      *
23      * @param from Address to sender
24      * @param to Address of recipient
25      * @param amount No of tokens to transfer (integer)
26      */
27     function transferFrom(address from, address to, uint256 amount) public returns (bool){
28         (bool success, bytes memory result) =
29             tokenAddress.call(abi.encodeWithSignature("transferFrom(address,address,uint256)",
30             from, to, amount));
31         require(success, "Call failed");
32         return abi.decode(result, (bool));
33     }
34 }
```

In line 17, the ThirdPartyTransfer contract sets the MyToken contract's address. Therefore, the MyToken contract should be deployed before the ThirdPartyTransfer. In lines 27-33, the `call` function is

used to perform the third-party token transfer. In this case, the third-party address is the ThirdPartyTransfer contract's address (once it is deployed), which should already have an allowance in the MyToken contract. Note how the `transferFrom` function signature is encoded into the ABI form.

Step 11. Compile the MyToken contract.

Return to `Deploy and run transaction` pane and select the first account on the `ACCOUNT` list. Deploy the MyToken contract to `Remix VM (Shanghai)`. Make sure to set the token name, symbol, and total supply. Note down the contract address.

Step 12. Compile the ThirdPartyTransfer contract.

Return to `Deploy and run transaction` pane and select the second account on the `ACCOUNT` list. Deploy the contract to `Remix VM (Shanghai)`. Make sure to enter the MyToken contract's address from Step 11 as the constructor parameter. Note down the contract address.

Step 13. On the `Deploy and run transaction` pane, select deployed MyToken contract. Copy the third pre-funded address from the `ACCOUNT` list (this account should be different from accounts used to deploy MyToken and ThirdPartyTransfer contracts). Change the `ACCOUNT` list back to the first account as total supply is initially assigned to the address that deployed the contract.

Transfer 1000 tokens to the address you just copied (once `to:` and `amount:` text boxes are filled, click on `transact` button). Make sure the third account has those 1000 tokens.

Step 14. Select the third account (i.e., the one you transferred tokens in Step 13) from the `ACCOUNT` list. Invoke the `approve` function after setting `spender:` to the ThirdPartyTransfer contract's address and an allowance `amount:` (set a value less than what you transferred, e.g., less than 1,000). Verify that the allowance is correctly set by invoking the `allowance` function by setting the owner and third-party addresses.

Step 15. On the `Deploy and run transaction` pane, select the deployed ThirdPartyTransfer contract. Setting the following values:

- `from:` – Third address that tokens were transferred to
- `to:` – Copy the fourth address from the `ACCOUNT` list and paste it here
- `amount` – Set the number of tokens to be less than the allowance

Change the account on the `ACCOUNT` list to the second address, i.e., address that deployed ThirdPartyTransfer contract. Invoke the `transferFrom` function. This transaction should be successful. Use `balanceOf` and `allowance` functions on MyToken to verify that account balances and allowance are correctly updated.

Step 16. Invoke the `transferFrom` function again while attempting to transfer same number of tokens as the initial allowance. This transaction should fail. Can you figure out why?