Instituto Superior Técnico, Universidade de Lisboa
Highly Dependable Systems

Smart Contract Execution Mechanisms

Important premise

This laboratory assignment covers the basic mechanisms of Solidity smart contract execution in the context of the Ethereum Virtual Machine that will be used throughout the course and the project. It is strongly recommended that all students get comfortable with this assignment before starting working on the project.

Goals

- Learn how to develop and compile Solidity smart contracts.
- Learn how to deploy and interact with EVM bytecode using the EVM provided by the Hyperledger Besu project.

1. Introduction

Smart contracts are self-executing computer programs deployed and operated within a blockchain environment. Since they run on blockchains, they function without the need for a central authority or server. Once deployed, a smart contract's code cannot be modified or updated due to the immutable nature of blockchains. However, it may include predefined functions that allow for data modifications. While information can be recorded in one block and removed in another, the historical record remains intact, ensuring transparency and auditability.

This assignment will be using the Solidity programming language to develop smart contracts. Solidity is a high-level, object-oriented programming language used for developing smart contracts. As a curly-bracket language, it uses { and } to define code blocks. Influenced by C++, Python, and JavaScript, Solidity is specifically designed to run on the Ethereum Virtual Machine (EVM). It is statically typed and offers features such as inheritance, libraries, and complex user-defined types.

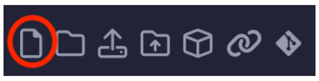
We will be using Remix to develop and compile our Solidity smart contracts. Remix is a web-based integrated development environment (IDE) for writing, compiling, deploying, and debugging Solidity code. It includes a built-in blockchain simulator called JavaScriptVM, which runs directly in your browser.

2. Hello World

In this exercise, you will learn how to build a “Hello World” smart contract with the smart contract development language Solidity. No prior knowledge is required.

Go to https://remix.ethereum.org to get started. Then follow these steps:

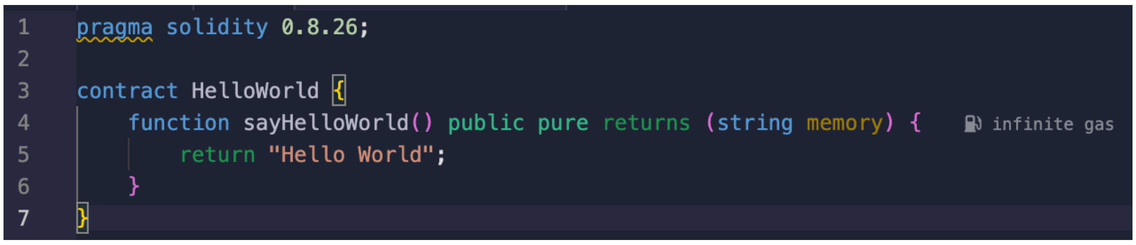
- Click on the button “Create new file” on the left bar called “File Explorer”:



- Give it the file name “HelloWorld.sol”. Files written in Solidity use the extension “.sol”:

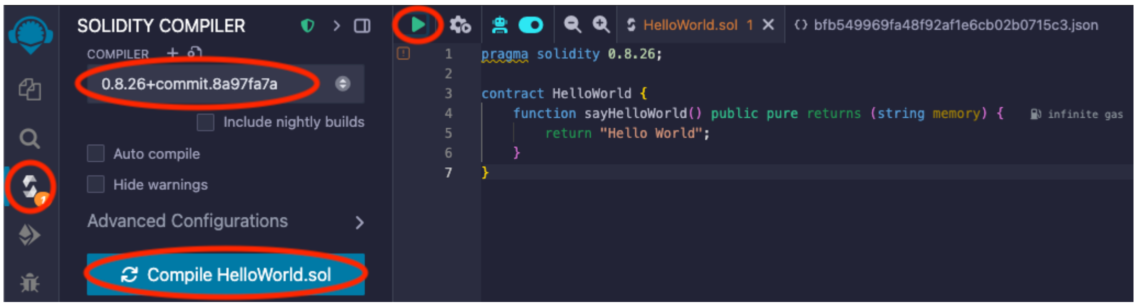
HelloWorld.sol

- Implement the code example bellow and save (e.g., ctrl + s or cmd + s):



```
pragma solidity 0.8.26;

contract HelloWorld {
    function sayHelloWorld() public pure returns (string memory) {    infinite gas
        return "Hello World";
    }
}
```

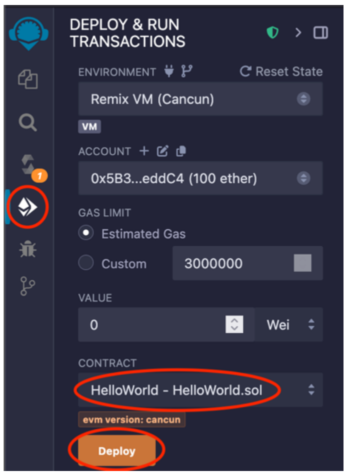
- To compile the smart contract click in the left side panel the button “Solidity Compiler”, select the compiler version as indicated in your “pragma” in your source code and hit the “Compile HelloWorld.sol” button. Alternatively, once the compiler version is correctly selected, you can click on the green play button at the top to perform a quick compilation of the source code:



0.8.26+commit.8a97fa7a

```
pragma solidity 0.8.26;

contract HelloWorld {
    function sayHelloWorld() public pure returns (string memory) {    infinite gas
        return "Hello World";
    }
}
```

- Next, we aim to deploy the smart contract locally in the browser using a JavaScript-based EVM by clicking in the left side panel the button “Deploy & Run Transactions”, selecting the “HelloWorld – HelloWorld.sol” contract, and hitting the deploy button:


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT — Reset State

Remix VM (Cancun)

VM

ACCOUNT

0x5B3...eddC4 (100 ether)

GAS LIMIT

● Estimated Gas

○ Custom 3000000

VALUE

0 Wei

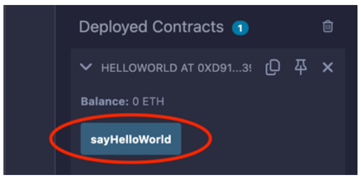
CONTRACT

HelloWorld - HelloWorld.sol

evm version: cancun

Deploy

- Now that the smart contract is deployed locally in our browser, we can interact with it and test it. To do so, scroll down the left side until you reach “Deployed Contracts”, expand the “HelloWorld”, click on the button that says “sayHelloWorld”:

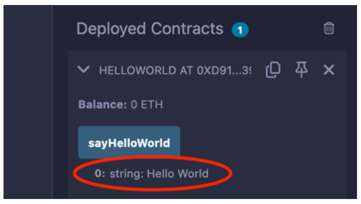

Deployed Contracts 1

HELLOWORLD AT 0XD91...3!

Balance: 0 ETH

sayHelloWorld

- This should return the message recorded in the smart contract: “Hello World”:


Deployed Contracts 1

HELLOWORLD AT 0XD91...3!

Balance: 0 ETH

sayHelloWorld

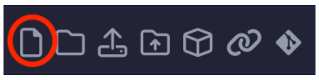
0: string: Hello World

3. Mappings and Storage

In this exercise, you will learn how to create and use mappings (i.e. HashMap in Java, dictionary in Python) to update and persist values across executions.

Go to https://remix.ethereum.org and follow these steps:

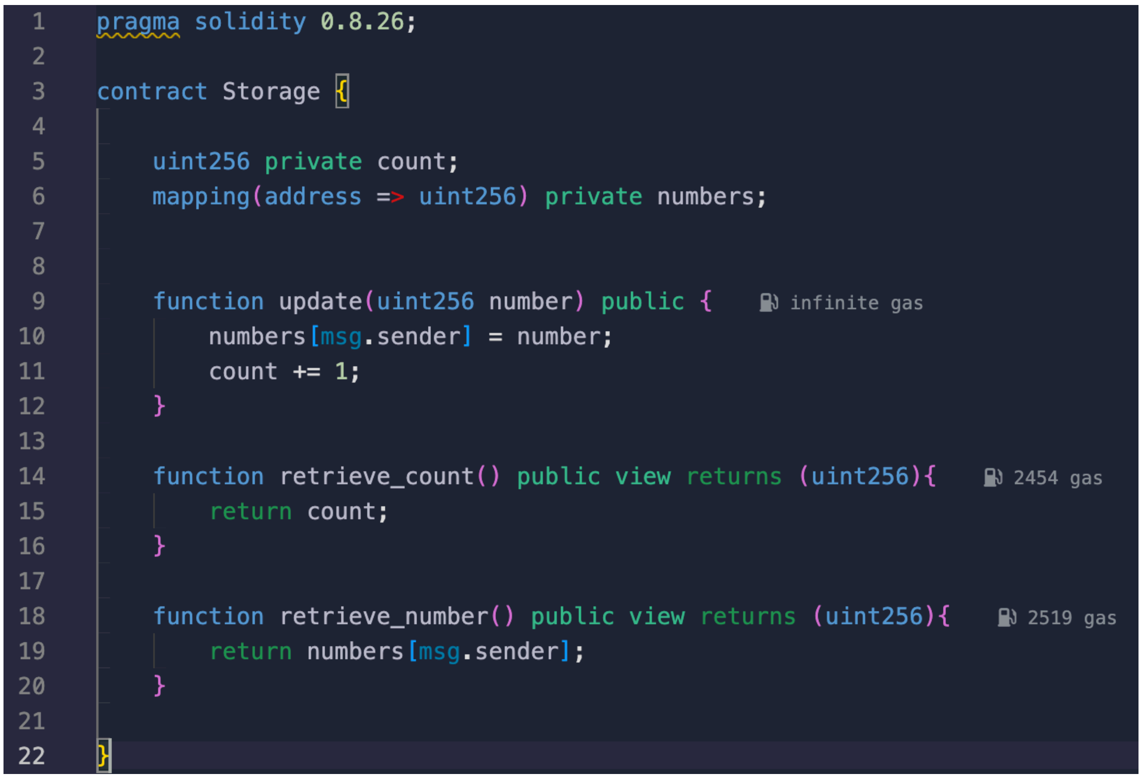
- Click on the button “Create new file” on the left bar called “File Explorer”:



- Give it the file name “Storage.sol”:

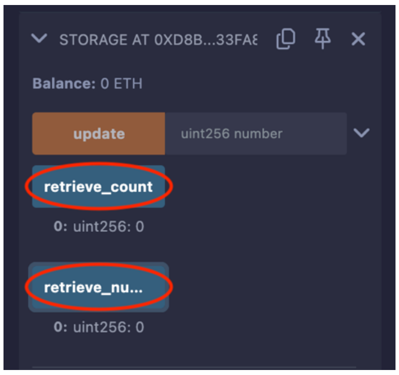
Storage.sol

- Implement the code example bellow and save it (e.g., ctrl + s or cmd + s). The example implements a mapping that keeps track of an unsigned integer value per user address. Please keep in mind that in Ethereum all global state variables (i.e., variables declared outside of functions are persisted through storage and not only accessible across functions but also across executions of transactions. Moreover, every value is by default always initialized with the value zero. This is the case for every type in Solidity (e.g., 0 for unsigned and signed integers, False for type bool, “0x0…0” for type address, etc.). The function “update” overrides the number associated to the calling address (i.e., msg.sender) with a new number originating from the input parameter called “number”, which is an unsigned integer of length 256 bit. Moreover, a global counter called “count” is incremented after every call to the function update to keep track of the total number of updates performed so far. The function “retrieve_count” returns the current value of the state variable count, while the function “retrieve_number” returns the current number that the calling user has stored:



```
pragma solidity 0.8.26;

contract Storage {

    uint256 private count;
    mapping(address => uint256) private numbers;


    function update(uint256 number) public {    infinite gas
        numbers[msg.sender] = number;
        count += 1;
    }

    function retrieve_count() public view returns (uint256){    2454 gas
        return count;
    }

    function retrieve_number() public view returns (uint256){    2519 gas
        return numbers[msg.sender];
    }

}
```

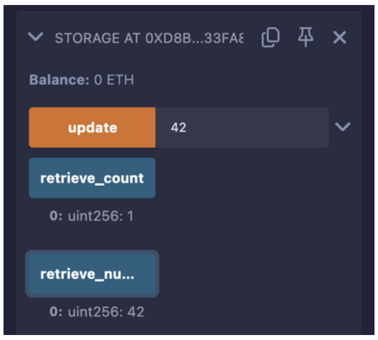
- Compile and deploy the smart contract locally as described in the earlier “HelloWorld” example. Once deployed, you can test the Storage smart contract by calling the “retrieve_count” and “retrieve_number” methods as shown below. Both function calls should return the default value zero as described earlier.



```
0: uint256: 0
```

```
0: uint256: 0
```

- Next, we can test if the update works correctly by calling the update function and passing the value “42” as the value for the parameter “number”. Afterwards, we can call again “retrieve_count” and “retrieve_number”, which should now return the value 1 for count and 42 for the number.



STORAGE AT 0XD8B...33FA8

Balance: 0 ETH

update 42

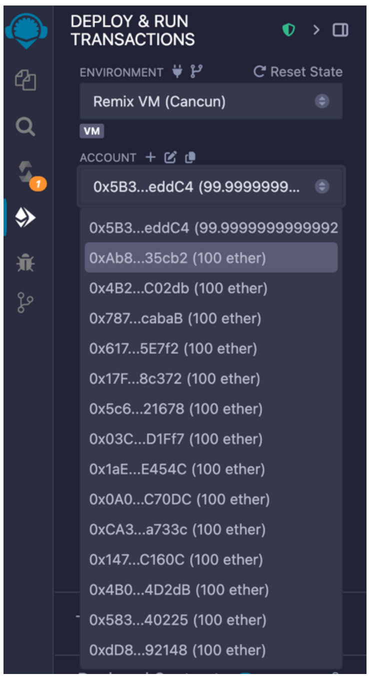
retrieve_count

0: uint256: 1

retrieve_nu...

0: uint256: 42

- Now keep in mind that a mapping is nothing else than a HashMap or a dictionary, meaning that we have a key and an associated value. In this example, the key is an address which is always the address of the calling account (i.e., msg.sender) according to our code. To test what number we get when using a different calling account we must select a different address in the “Account” selection at the top of “Deploy & Run Transactions”:



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

Reset State

Remix VM (Cancun)

VM

ACCOUNT

0x5B3...eddC4 (99.9999999...

0x5B3...eddC4 (99.9999999999992
0xAb8...35cb2 (100 ether)
0x4B2...C02db (100 ether)
0x787...cabaB (100 ether)
0x617...5E7f2 (100 ether)
0x17F...8c372 (100 ether)
0x5c6...21678 (100 ether)
0x03C...D1Ff7 (100 ether)
0x1aE...E454C (100 ether)
0x0A0...C70DC (100 ether)
0xCA3...a733c (100 ether)
0x147...C160C (100 ether)
0x4B0...4D2dB (100 ether)
0x583...40225 (100 ether)
0xdD8...92148 (100 ether)

- After selecting a different address, repeat the steps above with first calling the retrieve functions and then calling the update function and then again the retrieve functions. You should obtain initially value 1 for the counter and value zero for the number and after the update the counter should be incremented to 2 and the value returned by “retrieve_number” should be the same as you have provided as input to the function “update”.

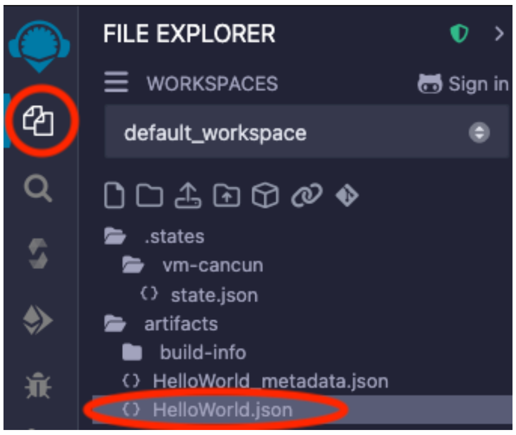
4. Hyperledger Besu's EVM Integration

In this exercise, you will learn how to extract the EVM bytecode from the previous examples in Remix, and how to deploy and interact with the EVM bytecode using Java implementation of the EVM provided by the Hyperledger Besu project.

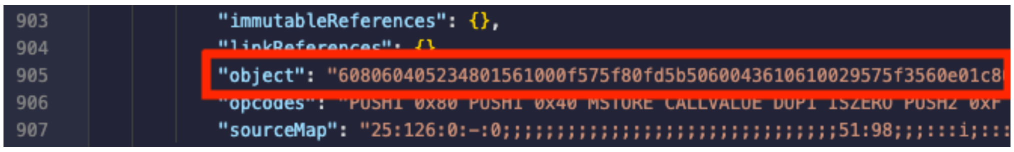
4.1 Deploy and Test 'HelloWorld' Example using Hyperledger Besu

Go to Remix and follow these steps to obtain the compile EVM bytecode from your HelloWorld smart contact:

- Open the "HelloWorld.json" file on the left bar of the "File Explorer" panel by expanding the "artifacts" folder. This file is generated by the Solidity compiler and contains several compilation information, included the EVM bytecode that we require:

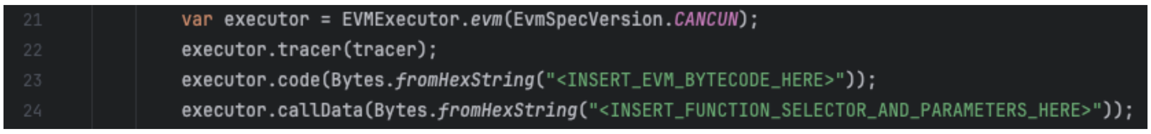


- Scroll totally down where is says "object". Copy the compiled EVM bytecode that you see starting with "60...". Please keep in mind that the file contains two "objects" with EVM bytecode. The first object contains the deployment EVM bytecode, however, we at this stage want to skip the deployment bytecode and directly obtain the "runtime" EVM bytecode. The deployment bytecode is executed only once and essentially includes the runtime bytecode which is copied to persistent storage and associated to a smart contract/account address. The deployment bytecode also includes any code and assignments that you might have included in the constructor of the smart contract.



```
"immutableReferences": {},
"linkReferences": {},
"object": "6080604052348015610010f575f80fd5b5060043610610029575f3560e01c8
"opcodes": "PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0xF
"sourceMap": "25:126:0:-:0;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;51:98;;;;::i;::
```

- Next, unzip "lab2.zip" and open the file "Main.java" contained within the folder "HelloWorld/src/main/java". Insert the copied EVM bytecode at line 23, where it says "<INSERT_EVM_BYTECODE_HERE>". We are passing the EVM bytecode of your HelloWorld smart contract to an instance of an EVM called "EVMExecutor", which as the name already suggests, executes EVM bytecode:

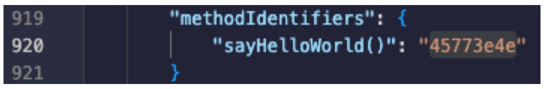


```
21            var executor = EVMExecutor.evm(EvmSpecVersion.CANCUN);
22            executor.tracer(tracer);
23            executor.code(Bytes.fromHexString("<INSERT_EVM_BYTECODE_HERE>"));
24            executor.callData(Bytes.fromHexString("<INSERT_FUNCTION_SELECTOR_AND_PARAMETERS_HERE>"));
```

- The EVMExecutor can also read inputs. For instance, we need to tell the EVM what smart contract function we want to execute. This is done via the input data which is given via the callData() function at line 24. We select a smart contract function by providing a so-called function selector which is computed as the first 4 bytes of the Keccak256 hash of the function signature. The function signature is composed of the function name and the parameter types that the function has. In our case the function signature for "sayHelloWorld" is 'sayHelloWorld()' since it has no parameters. The Keccak256 of the function signature is then:

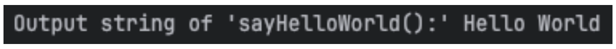
  45773e4ecd15c37fc2d477436760690867b48c8935e4278061c48977dddf5e38

- You can compute it yourself quickly via the following website: https://emn178.github.io/online-tools/keccak_256.html and using the function signature above. The final function selector is the first 4 bytes of the hash, thus "45773e4e". Insert this function selector into the callData function at line 24. You can also find the function selector within Remix by scrolling further down within the file "HelloWorld.json" and looking at the "methodIdentifiers":



```
    "methodIdentifiers": {
        "sayHelloWorld()": "45773e4e"
    }
```

- After inserting the EVM bytecode and the function selector, run the file and verify that your output matches the output below:

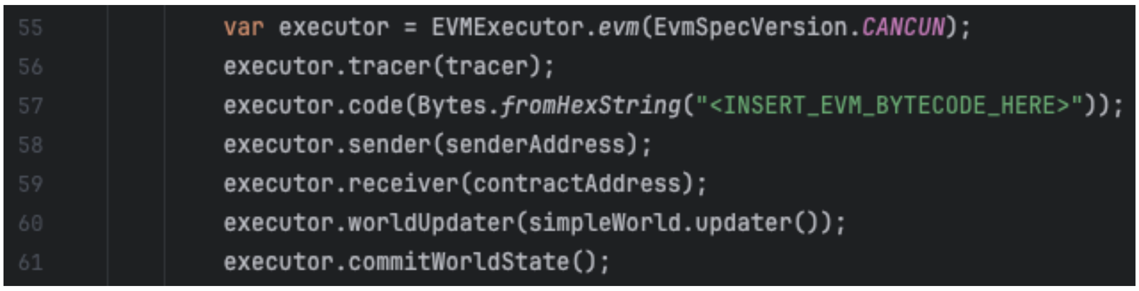


```
Output string of 'sayHelloWorld():' Hello World
```

- The code will execute the EVM bytecode and keep track of every instruction that has been executed using a "tracer". The execution will stop in this case after executing the EVM instruction "RETURN". We can extract our "Hello World" return string by extracting the offset and size from the stack and using this information to extract the return data from the memory. Afterwards we need to parse the memory to obtain the string in plaintext. The first 32 bytes denote offset within the return data where the length of the string is stored. This is typically at offset 32, hence reading the next 32 bytes from the return data will give us the length. After knowing the length, we can extract the string encoded as UTF-8 starting from the next 32 bytes up until the length that we extracted. Luckily, the function "extractStringFromReturnData" does all of this already for us.

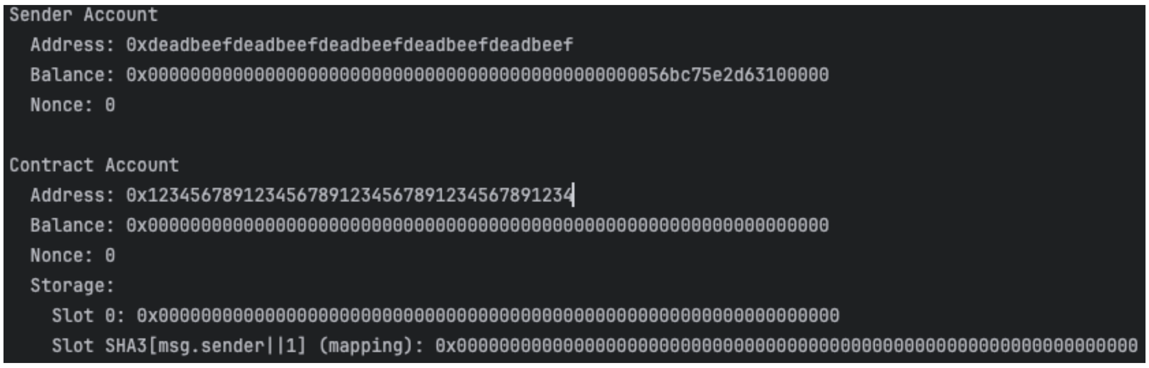
4.2 Deploy and Test 'Storage' Example using Hyperledger Besu

- Go to Remix and follow the same instruction from the “HelloWorld” example to extract the EVM bytecode and insert it into the Main.java file contained in the folder “Storage/src/main/java”:



```java
        var executor = EVMExecutor.evm(EvmSpecVersion.CANCUN);
        executor.tracer(tracer);
        executor.code(Bytes.fromHexString("<INSERT_EVM_BYTECODE_HERE>"));
        executor.sender(senderAddress);
        executor.receiver(contractAddress);
        executor.worldUpdater(simpleWorld.updater());
        executor.commitWorldState();
```

- The code already includes all function selectors. The first part of the code creates a world state, that keeps track of the state of every account on the blockchain. Two accounts are created, a sender account with address “deadbeefdeadbeefdeadbeefdeadbeefdeadbeef” and a contract account with address “1234567891234567891234567891234567891234”. The sender’s role is to act as the sender of a transaction and hence the triggering account of an execution of a transaction (i.e., msg.sender in Solidity). The contract account reflects the smart contract deployed on the blockchain and besides also containing a balance, also contains a key-value store called storage. After running the code, the initial part of the output should look like this:



```
Sender Account
  Address: 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
  Balance: 0x0000000000000000000000000000000000000000000000056bc75e2d63100000
  Nonce: 0

Contract Account
  Address: 0x1234567891234567891234567891234567891234
  Balance: 0x0000000000000000000000000000000000000000000000000000000000000000
  Nonce: 0
  Storage:
    Slot 0: 0x0000000000000000000000000000000000000000000000000000000000000000
    Slot SHA3[msg.sender||1] (mapping): 0x0000000000000000000000000000000000000000000000000000000000000000
```

Two storage slots (i.e., keys) are printed for the contract account. Slot “0” is associated to the global state variable “count” in our Solidity smart contract as it is the first state variable. The second storage slot represents the slot where the number for msg.sender is stored. This slot is computed as the Keccak 256 hash of the sender address (padded with leading zeros to be 256 bit large) with the value 1 (padded with leading zeros to be 256 bit large). The value 1 originates from the fact that in our Solidity smart contract our mapping is the second state variable, hence it is placed at index 1.

- Next, the code makes calls to the functions “retrieve_count()” and “retrieve_number()”, both returning value 0 as this is the default value of each uninitialized variable in Solidity. Then, the code makes a call to the function “update()” passing the value 42 as parameter. The value 42 is converted to hex representation and padded with leading zeros to a achieve a length of 256 bit before being passed to the EVM as input for execution. Afterwards, the code calls the functions “retrieve_count()” and “retrieve_number()” and, but this time “retrieve_count()” returns the value 1 while “retrieve_number()” returns the value 42. You should see and output like this:

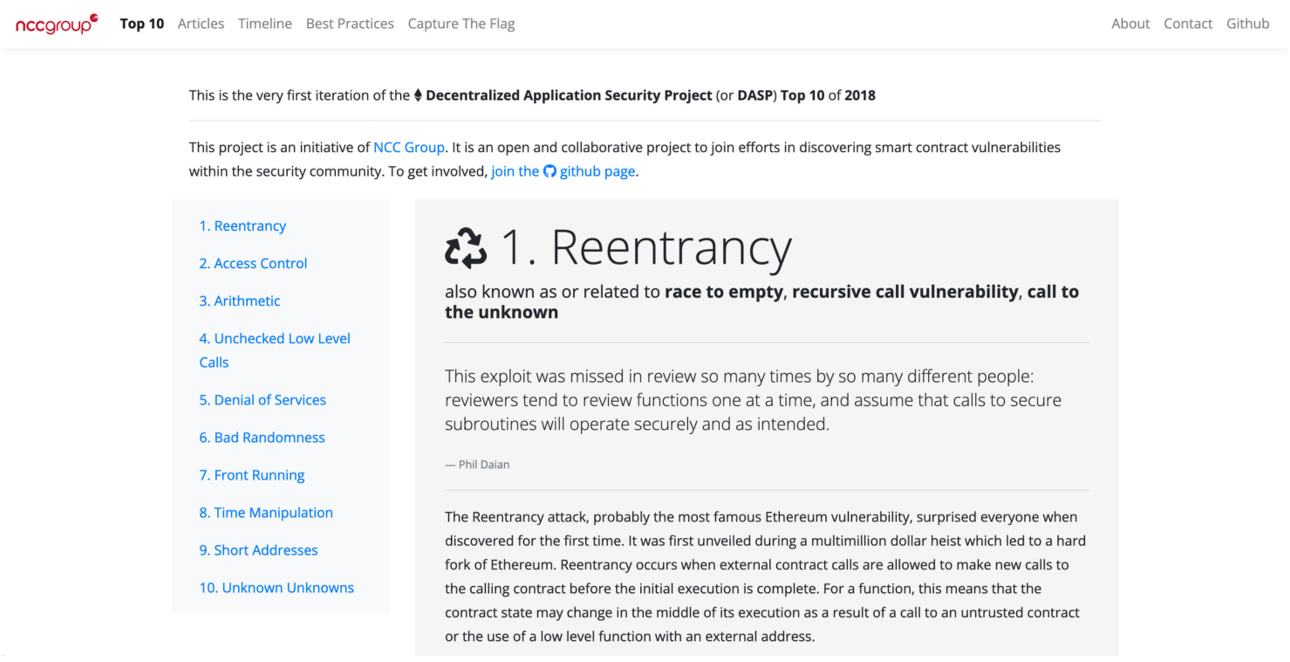
```
Output of 'retrieve_count():' 0
Output of 'retrieve_number():' 0
Output of 'retrieve_count():' 1
Output of 'retrieve_number():' 42
```

- Finally, the code outputs once again the state of both accounts. For the contract account we observer that the Slot 0 value has changed to 0x1 and that the value of the mapping slot has changed to 0x2a, which is hexadecimal for 42:

```
Sender Account
  Address: 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
  Balance: 0x00000000000000000000000000000000000000000000000056bc75e2d63100000
  Nonce: 0

Contract Account
  Address: 0x1234567891234567891234567891234567891234
  Balance: 0x0000000000000000000000000000000000000000000000000000000000000000
  Nonce: 0
  Storage:
    Slot 0 (count): 0x0000000000000000000000000000000000000000000000000000000000000001
    Slot SHA3[msg.sender||1] (mapping): 0x000000000000000000000000000000000000000000000000000000000000002a
```

5. Smart Contract Security

If you are interested in learning more about developing smart contracts in Solidity and at the same time learn more about the security issues that can arise from developing smart contracts, please take a look at examples available at https://dasp.co/:



This is the very first iteration of the **Decentralized Application Security Project** (or **DASP**) **Top 10** of **2018**

This project is an initiative of NCC Group. It is an open and collaborative project to join efforts in discovering smart contract vulnerabilities within the security community. To get involved, join the github page.

1. Reentrancy
2. Access Control
3. Arithmetic
4. Unchecked Low Level Calls
5. Denial of Services
6. Bad Randomness
7. Front Running
8. Time Manipulation
9. Short Addresses
10. Unknown Unknowns

1. Reentrancy

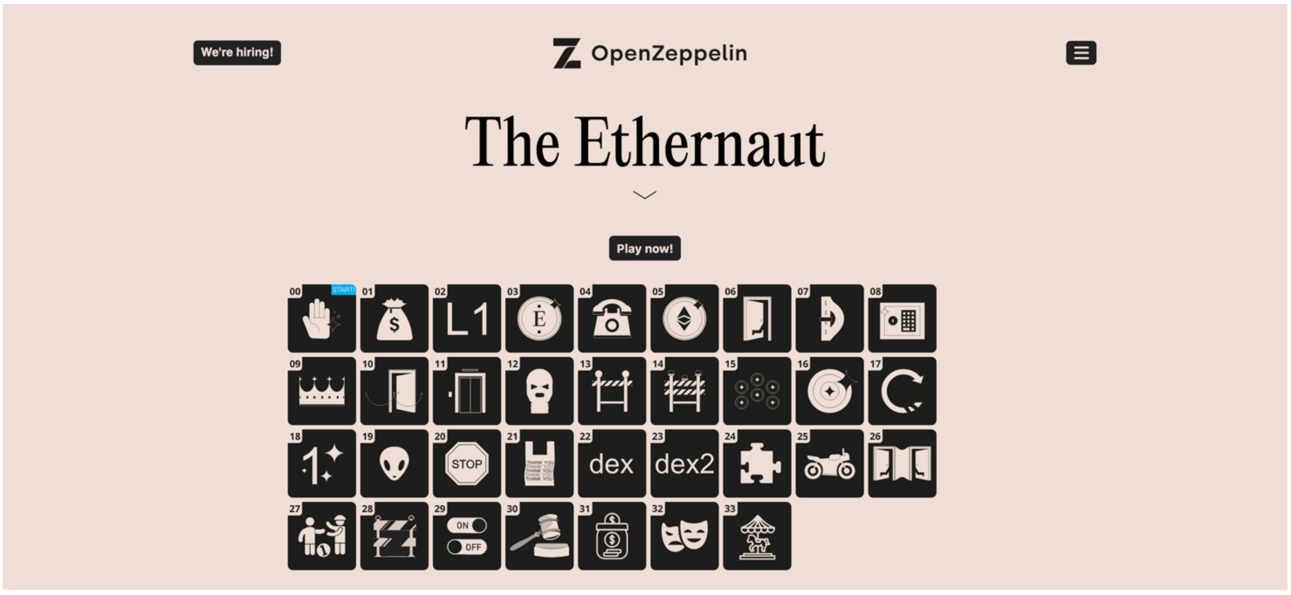
also known as or related to **race to empty**, **recursive call vulnerability**, **call to the unknown**

This exploit was missed in review so many times by so many different people: reviewers tend to review functions one at a time, and assume that calls to secure subroutines will operate securely and as intended.

— Phil Daian

The Reentrancy attack, probably the most famous Ethereum vulnerability, surprised everyone when discovered for the first time. It was first unveiled during a multimillion dollar heist which led to a hard fork of Ethereum. Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

and also try to solve the different challenges available on https://ethernaut.openzeppelin.com/:



a CTF-alike challenge for exploiting vulnerabilities in smart contract with different levels of difficulty.