

# Tutorial 3: Smart contracts as autonomous and self-enforcing agents

Patrick McCorry

Kings College London, UK  
patrick.mccorry@kcl.ac.uk

**Abstract.** An introduction into how smart contracts can interact with other smart contracts. We'll cover how to define interfaces to help interact with other smart contracts via a simple Oracle contract, how the fallback function works, and try out the re-entrancy attack that took down TheDAO. It is recommended to make notes on this tutorial sheet for future use.

## 1 A platform of autonomous and self-enforcing agents

A core vision for a global and public blockchain network like Ethereum is to provide a platform for autonomous and self-enforcing agents. Once deployed a smart contract becomes an autonomous agent that can act on the owner's behalf, but potentially without any future interference. This has led to the community designing decentralised autonomous organisations (and applications) like TheDAO, the Ethereum Name Service and publicly verifiable Ponzi schemes like FOMO3D. To push forward this dystopian future, we'll explore how the EVM facilitates contract to contract interaction. By the end of this tutorial, you'll learn how to build a smart contract factory, a simple Oracle contract and how to break an existing smart contract via a reentrancy attack.

## 2 Contract Interaction in Ethereum

We'll provide an overview of defining a contract's interface, how a contract can invoke another contract, and how a contract can instantiate new contracts. This will bring us to the fallback function (introduced in Tutorial 1) which is invoked if a caller tries to interact with an undefined function. Afterwards we'll briefly explore some special opcodes to support more advanced features.

### 2.1 Interaction Overview

We'll need to consider the following three questions to understand how contract A can communicate with contract B:

- How can contract A reference (or instantiate) contract B?
- What is the syntax for contract A to invoke a function in contract B?
- How does contract A know that contract B implements the function?

We'll answer each question in turn.

*Contract Interface* An interface provides a list of function definitions (i.e. name, parameters, return list, etc) without an implementation. Contract A can include this interface before it is deployed, and it will expect contract B to implement all functions in the interface. The difference when defining an interface (instead of a contract) is that each function's definition ends with ';' instead of '}'. We provide an example in Figure 1.

```
Contract Interface

contract ContractInterfaceExample {

    function function1() public;
    function function2(uint counter) public;
    function function3() public view returns(bool);

}
```

Fig. 1: How to define a contract interface

*Interacting with other contracts* Before we can discuss contract interaction, contract A must have a reference to an instance of contract B on the network. We have two options:

- Contract B already exists on the network and we can simply use its address,
- Contract A can create a new instance of contract B.

If we assume C is the contract's interface name, then we can store a reference to an already existing contract by:

```
C public name = C(address con)
```

On the other hand, let's say instead of a contract interface, we include a full implementation of another contract. For example in Figure 2:

```
Contract Interface

contract CImpl {

    function function1() public { }
    // Some logic is implemented
}

}
```

Fig. 2: How to define a contract interface

Then we can create a new instance of an implemented contract by:

```
CImpl public name = new CImpl()
```

Once we have an address for contract B on the network, then we can simply invoke a function with `name.functionName()`, where `name` is the variable and `functionName()` is a function defined in the interface. A full example of defining an interface, referencing another smart contract and invoking a function is provided in Figure 3:

```
Overview of creating and interacting with other contracts

contract BadRNG {
    // We don't need to define the function
    function getRNG() public returns(uint);
}

contract GameLog {
    string[] logs;

    // We can fully implement the function
    function updateLog(string memory msg) public {
        logs.push(msg);
    }
}

contract GamblingGame {
    BadRNG public badRNGcon;
    GameLog public logcon;

    constructor (address _badRNG) {
        // Reference external contract using supplied address
        badRNGcon = BadRNG(_badRNG);
        // Instantiate new instance of GameLog
        logcon = new GameLog();
    }

    function play(string memory _winnerinfo) public {
        // Some game logic
        uint num = badRNGcon.getRNG();
        // Use num in game
        logcon.updateLog(_winnerinfo);
    }
}
}
```

Fig. 3: How to reference, instantiate and interact with another contract

*Trusting other contracts (and the fallback function)* If contract A is responsible for instantiating contract B, then it can trust the correct code was deployed. However if contract A is only provided the address of contract B, and contract B claims to implement the contract interface, then the final question is whether contract A can verify contract B has *actually implemented* the interface. This brings us to the following example:

```
Contract Interface

contract Coffee {
    function sip() public;
    function getSips() public returns(uint);
}

contract NotRelated {
    uint public c;
    function () external { c = c + 1; }
}

contract Alice {
    Coffee public coffee;

    constructor(address _coffee) public {
        coffee = Coffee(_coffee);
    }

    function tryCoffee() public {
        coffee.sip();
    }
}
```

Fig. 4: How to define a contract interface

Let's imagine a scenario where `_coffee` is the contract address for `NotRelated`, and it is used to set up `Alice` contract. Then what happens when a user calls `Alice.tryCoffee()`? Take this opportunity to try and work it out for yourself. Please make some notes on this tutorial sheet before turning to the next page.

Answer: The fallback function `function () external` in `NotRelated` is executed and the counter `c` is incremented by one.

Key Point: A contract cannot verify that another contract has actually implemented the interface. Instead of throwing an exception and terminating the execution, the fallback function will always be executed. As we'll see later, this can have devastating implications.

*Special opcodes* So far we have presented how to define a contract interface and how to interact with other functions. The underlying EVM provides special opcodes that lets us dynamically select a function in another contract (and thus we do not need to define an interface altogether):

- `call` I'm contract D and I want to execute contract E's function for contract E's storage. *Typical object-oriented programming. Calling `E.function()` can only update E's storage.*
- `callcode` I'm contract D and I want to execute contract E's function for my storage. *Contract D is essentially copying E's function. Calling `E.function()` will update D's storage.*
- `delegatecode` I'm a contract D and I want to execute contract E's function, but contract E can pretend to be me. *Contract D is essentially delegating the function call to E. E's function can overwrite D's storage and pretend to be D to any other contract. i.e. `msg.sender` will be D's address, not E.*

The above opcodes brings us to the EVM assembly language which is out-of-scope for our tutorials. Someday if you want to be a real smart contract developer, it is crucial to understand what is happening at the assembly level. So we encourage interested students (i.e. you) to check online resources for more information.<sup>1</sup>

---

<sup>1</sup> A good starting point is here <https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall>

### 3 Exercise: Building an Oracle Contract

Today's task is to build a simple oracle contract that collects information from the outside world and provides it for easy access to other contracts on the network. We'll provide a contract interface for the oracle and a simple contract that will request information from the oracle. It is your job to implement the oracle according to the interface, and let the contract use your data.

#### 3.1 Sample Contract

In Figure 5, we provide a contract interface for the oracle and a simple contract that will request information from the oracle. It is your job to implement the oracle according to this interface, and let the contract use your data.

```
Oracle Exercise

contract FootballOracle {
  // All matches are indexed. Returns whether query was successful,
  // alongside the scores
  function getScore(uint matchid) public
    returns(bool success, uint score1, uint score2);
}

contract EmitMatchEvent {
  event MatchScore(uint matchid, uint score1, uint score2);
  FootballOracle public oracle;

  constructor(address _oracle) public {
    oracle = FootballOracle(_oracle);  }

  function checkScore(uint _matchid) public {
    bool success; uint score1; uint score2;
    // Fetch scores from the oracle
    (success, score1, score2) = oracle.getScore(_matchid);

    // If query works, tell world about the score!
    if(success) {
      emit MatchScore(_matchid, score1, score2);
    }
  }
}
}
```

Fig. 5: Task: Implement the Football Oracle and let EmitMatchEvent use it!

#### 3.2 What to consider for the Football Oracle Task

Try to consider the following in your solution:

- How will the FootballOracle accept new football results?
- Do you authenticate who is sending information to the oracle? Are they trusted?
- How are the football results stored in the contract? Can they be removed after an expiry period?
- Is there a good method for us to trust the code deployed at `_oracle` which is used for `EmitMatchEvent`?

## 4 Fun Exercise: Reentrancy attack

There are two components for the reentrancy attack.

- *Reentrant contracts.* A smart contract with recursive functions, where functions can reference and re-call themselves.
- *Fallback function.* An unnamed function which is always executed when the caller tries to invoke a function that doesn't exist.

We'll explore both components separately before letting you try out the reentrancy attack that infamously drained over \$150m from TheDAO.

### 4.1 Contract Recursion

*Recursive Functions* Solidity (and the EVM) supports recursive functions, where a function can reference and re-call itself several times. Typically recursion has a *base case* that implies *when we reach this case, stop all execution* and a *induction step* that implies *we haven't reached the base case, so lets re-execute the function again*. Let's consider a simple example in Figure 6.

```

Recursively count to 10
contract RecursionInContract {
  uint c = 0;
  function countTo10() public {
    if(c == 10) { // Base case
      return;
    } else {
      c = c + 1; // Inductive step
      this.countTo10();
    }
  }
}

```

Fig. 6: Recursive Function

As the example clearly illustrates, `RecursionInContract.countTo10()` has a base case to check if `c==10` and an inductive step to increment `c` by 1. Every

time the contract invokes `countTo10()`, it first checks the base case is satisfied. If `c != 0`, then `c` is incremented by one and the function `countTo10()` is re-invoked. The recursion stops when the base case is reached by returning (i.e. exiting the function call). All other calls will simply finish the function's execution.

*Recursive Contracts* While it may appear strange at first, this recursive behaviour can be extended to let two contract's re-call each other until a base case in one of the contract's is hit. Again, let's consider a simple example:

```
Caller recursively calls Counter to count to 10

contract Caller {
    function repeat(address _counter) public;
}

contract CallerImpl {
    function repeat(address _counter) public {
        Counter(_counter).countTo10();
    }
}

contract Counter {
    function countTo10() public;
}

contract CounterImpl {
    uint256 c;

    function countTo10() public {
        if(c==10) {
            return;
        } else {
            c = c + 1;
            Caller(msg.sender).repeat(address(this));
        }
    }
}
```

Fig. 7: Recursive Contracts

Let's break down Figure 7. There are two contracts:

- *CallerImpl* will always try and invoke the function `Counter.countTo10()` for the supplied address `_counter`.



- *CounterImpl* has a single function, `countTo10()`, that checks if the counter is 10. If it isn't, then it will ask the *Caller* to re-invoke this function.

To trigger the recursion, a user must call `CallerImpl.repeat()` with the `CounterImpl`'s address. This triggers the following loop:

- `CallerImpl` invokes `CounterImpl.countTo10()`,
- `CounterImpl` re-invokes `CallerImpl.repeat()`,

Every time `CounterImpl.countToTen()` is invoked, it'll first check the base case (i.e. does `c==10`?) before moving onto the induction step which increments `c` by 1. Eventually the base case will be hit (i.e. `c == 10`). Similar to before, `CounterImpl.countTo10()` simply returns. This breaks the loop as `CallerImpl` is not re-invoked. All previous invocations/calls can finally finish their execution.

**Don't worry if this part doesn't make sense right away. Try plugging the code into remix and running the debugger step by step.**

## 4.2 Fallback function

The next component we need to consider before deploying the attack is the fallback function. It was originally designed for two purposes:

- *Coin-management* If coins are sent to a smart contract and no function is specified, then the fallback function is used to accept the coins in a meaningful manner.
- *Error-handling* If a user (or another smart contract) calls a function that doesn't exist in the smart contract, then the situation can be gracefully handled.

The coin-management purpose is desirable as it lets transferring coins to a smart contract be exactly the same as an Ethereum account (i.e. indistinguishable). When the coins are sent to the smart contract, the fallback function can accept the coins and allocate it in a meaningful way for the program. However as we'll see in the next section, it can have devastating implications.

## 4.3 Try it yourself! Take down this contract

Before we go into the details of how to combine both components to perform a re-entrancy attack, there is a simple lesson we should learn about secure protocol design.

*The 'told you so' thread* Back in 2014, Andrew Miller posted about reentrant contracts and he was concerned that letting a contract recursively call another contract looked dangerous. He proposed Ethereum should incorporate a mutex and not let a contract be re-invoked until it has finished its current execution.<sup>2</sup> The concern was treated in the following way:

---

<sup>2</sup> <https://forum.ethereum.org/discussion/1317/reentrant-contracts>

- gavofyork: What problem, in the simplest case, are you trying to solve?
- socrates1024: To allow for simple security reasoning about contracts that contain outgoing calls to other contract, it's useful to have a way to "schedule" the call to occur after the current transaction completes.

The conversation ended with that message, but two years later the exact recursive mechanism (alongside the fallback function) was used to drain over \$150m coins from TheDAO. The key takeaway here are the proverbs *complexity is the enemy of security* and *attacks only get better*. If there is a 40-line program holding \$100m+ assets, we as a community must foster an execution environment that supports straight-forward security reasoning.

*How does reentrancy work?* The `VictimContract` has a `withdraw` function that will send coins to `msg.sender` (i.e. the function's caller). There are three issues that will make this `withdraw` function vulnerable:

- *Interaction before effect.* The `withdraw` function will send coins to `msg.sender` (i.e. the function's caller) before deducting their balance internally.
- *Invoking the fallback.* The `withdraw` function will always invoke the attacker contract's fallback function (i.e. this is expected behaviour).
- *Transfers too much gas.* All remaining gas for this transaction is transferred to the attacker contract's fallback function.

The `AttackerContract`'s fallback function is designed to do the following:

- *Base case check.* Does the `VictimContract` have any coins remaining? (i.e. `VictimContract.balance == 0?`).
- *Inductive step.* If the `VictimContract` still has coins, then re-call its `withdraw` function. Otherwise do nothing.

As you will see in the next section, the above conditions ensure that every time `VictimContract.withdraw()` is re-invoked by `AttackerContract`, the attacker always has a balance that can be withdrawn. The attack is finished after the attacker has stolen all coins from the victim.

*Its your turn to try it for real* Figure 8 provides the broken `VictimContract` and an unfinished `AttackerContract`. It is your duty to fill in the blanks and deploy the attack. Here are some tips to try it out in Remix:

- Deploy the `VictimContract` and use `deposit()` to send 10 ether.
- Copy the `VictimContract`'s address and use it when deploying the `AttackerContract`.
- Run `AttackerContract.attack()`.
- Use `getAttackerBalance()` and `getVictimBalance()` to check if the attack was successful!

Fill in the blanks and try out the attack

```
contract VictimContractInterface {
    function withdraw() public payable;
}

contract VictimContract {
    uint256 toTransfer = 1 ether;

    Only 1 ether can be sent by this contract
    function withdraw() public payable {
        // Send 1 coin
        msg.sender.call.value(toTransfer)("");
        // Deduct balance by 1
        toTransfer = 0;
    }

    // Use deposit() to send 10 ether to contract
    function deposit() public payable {};
}

contract AttackerContract {
    address victim;

    constructor(address _victim) public {
        victim = _victim;
    }
    // Trigger the attack
    function attack() public payable {
        // Fill in the blanks
    }
    function () external payable {
        // Fill in the blanks
    }
    function getAttackerBalance() public view returns(uint) {
        return address(this).balance;
    }
    function getVictimBalance() public view returns(uint) {
        return address(victim).balance;
    }
}
```

Fig. 8: Fill in the blanks!

#### 4.4 How do we prevent the reentrancy attack

We'll summarise some of the key take aways on why the above attack works and what we can do to mitigate it.

*Be careful with allocating gas to external contracts* While we have not yet mentioned it, the code `msg.sender.call.value(toTransfer)("")` automatically transferred all the remaining gas to the AttackerContract every time it was invoked. One way to prevent the above attack is to simply restrict the gas transferred to another contract when it is invoked. For example, `transfer` and `send` only transfers 23k gas to the contract receiving coins (i.e. AttackerContract). This isn't enough gas for recursion and thus stops the attack.

*Checks-effects-interaction paradigm* TheDAO hack encouraged a new programming heuristic to avoid this style of attack in the future:

- *Checks*. Check all preconditions before executing any code (i.e. does the caller have a sufficient ballance?)
- *Effects*. Update all internal state (i.e. pretend that the transfer has already happened)
- *Interactions*. Interact with other smart contracts (and revert all computation if there is a failure). (i.e. transfer coins)

Try implementing the above heuristic for our broken contract and see for yourself if the reentrancy attack will still work!