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

Patrick McCorry

Cryptocurrency Class 2022
`stonecoldpat@gmail.com`

Abstract. We cover how a smart contract can interact with another smart contract alongside a special attack exercise that took down the original DAO contract. It is recommended to make notes on this tutorial sheet for future use.

1 A platform of autonomous and self-enforcing agents

A platform like Ethereum has hundreds of autonomous and self-enforcing agents whom are implemented as smart contracts. This makes up the majority of DeFi applications like Uniswap who can operate without human interference. There are other DeFi Applications like MakerDAO that do rely on information from the real-world via oracle contracts, but the goal is to define what requires and does not require human trust in the application. To facilitate a world of smart contracts, it must be possible for a smart contract to interact with another smart contract. This is the focus of our tutorial and we cover the following:

– *Contract interaction.* How a smart contract can invoke a function in another smart contract.
– *Oracle contract.* How to build a smart contract that provides information about the real-world.
– *Re-entrancy attack.* A coding exercise on how build a malicious smart contract that can exploit another vulnerable smart contract.

2 Contract Interaction in Ethereum

We consider the following three questions:

– 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?

2.1 Interaction Overview

Contract Interface We provide an example in Figure 1. An interface provides a list of function signatures without an implementation. This allows contract A to invoke functions in another smart contract that is compatible with the interface.

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

```
LittlePony public name = LittlePony(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 LittlePonyImpl {

  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:

```
LittlePonyImpl public name = new LittlePonyImpl()
```

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. It does not throw an exception or terminate the execution.

Special opcodes The EVM has special opcodes for invoking another smart contract and it can allow us to dynamically pick which function is executed. This allows us to invoke another smart contract without implementing an interface in advance. There are three opcodes:[1]

- `call` Contract A calls a function in Contract B. The execution happens in the context of Contract B and it updates the storage of contract B.
- `delegatecode` Contract A calls a function in Contract B. The execution happens in the context of Contract A and it updates the storage of contract A.
- `staticcall` Contract A calls a function on Contract B. The execution allows for read operations, but the state of Contract B cannot be changed.

A *call* is similar to calling a function in an object-oriented language. The other object is executed and it returns back to the caller. The *delegatecall* is exotic as it allows a contract to adopt new functionality that is applied to its storage. It is a bit like dynamically importing a library to include new functionality and in this case a smart contract invokes the other smart contract to benefit from new functionality. [2]

[1] https://docs.soliditylang.org/en/v0.8.11/050-breaking-changes.html?highlight=delegatecall#functions
[2] 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 provide a contract interface for the oracle and a simple contract that requests information from the oracle. It is up to you to implement an oracle according to the interface and let the simple 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 seperately 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 satisified. 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 beheaviour 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 chase (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.[3] The concern was treated in the following way:

[3] 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 fuction) was used to drain over $150m coins from TheDAO. The key takeaway here are the provebs *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 depost() 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 Let's consider the code of calling into another smart contract:

msg.sender.call{value: 10000 gas: 3000}('0x')

If the gas field is left blank, then it transfers 63/64[4] of the remaining gas to the AttackerContract which provides sufficient gas to perform the re-entrancy attack. The attack can be prevented by restricting the gas available to the attacker. 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 it can stops the attack... but a recent upgrade broke this protection for some smart contracts.[5]

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! Please make some notes on why the attack now fails.

[4] https://eips.ethereum.org/EIPS/eip-150
[5] https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/