Tutorial 2: Handling Payments

Patrick McCorry

Cryptocurrency Class 2022
stonecoldpat@gmail.com

Abstract. An introduction into how to handle the deposit and withdrawal of *ether* in a smart contract. We'll cover environmental variables, basic cryptographic operations, and finally the check-effects-interaction paradigm. It is recommended to make notes on this tutorial sheet for future use.

1 Special Features in Smart Contracts

Every smart contract lives within the Ethereum Virtual Machine (EVM). It has a unique address, a balance in ETH, local storage and its code is publicly available. A smart contract is treated the same as an Ethereum account for sending and receiving ether, but the benefit is that a developer can set a list of conditions that must be satisified before the ether is sent. This programmability is a cornerstone of what makes a system like Ethereum so interesting to study as it can self-enforce a set of custom rules before allowing a user to update the database. With that backdrop, we will cover the following in this article:

- **Environmental variables.** Allows a smart contract to self-inspect the blockchain's status, the immediate caller, and the shared global clock.
- **Basic cryptographic operations.** How to use hash functions and verify digital signatures.
- **Pre-conditions.** How to set conditions that are checked during execution to determine whether the transaction was successful or failed).
- **Deposit and refunds.** A coding exercise on how to handle payments in a smart contract.

1.1 Environmental Variables

We'll only consider a subset of the data a smart contract can inspect about its environment. This includes the invocation transaction, the message that accompanies invoking a transaction, the state of an address on the network and the blockchain.

Transaction **tx** lets us access information about the transaction that is invoking the smart contract and it has two values:

- **tx.gasprice** or **gasleft**() specifies the gas price chosen by the transaction caller.
- **tx.origin** specifies the Ethereum account of the user who signed the transaction.

Message **msg** lets us access information about the immediate caller of the smart contract's function. We'll consider the following values:

- **msg.sender** specifies the address of the function's immediate caller.
- **msg.value** specifies the number of ether being sent to this smart contract.

Note the message reflects information about the immediate caller of this function including the number of coins sent.

Address **address** lets us access information about an address on the network. For example, it is declared:

address admin = '0x...';

We can fetch information about an account in the database:

- **address.balance** is the address' balance.
- **address.transfer**(**uint** `amount`) transfers `amount` ether to the address from this smart contract. An exception is thrown if it fails and all computation so far is reverted.
- **address.send**(**uint** `amount`) **returns**(**bool**) transfers `amount` ether to the address from this smart contract. It returns whether it is successful with TRUE or FALSE.

You can use **this** to fetch the address of the current smart contract. There are some tricks to distinguish if an **address** is a smart contract or an Ethereum account. We'll leave that as a homework exercise, but generally speaking, the goal is to make it indistinguishable.

Blockchain **block** lets us access information about the blockchain and we'll consider the following values:

- **block.coinbase** the miner of this block's address.
- **block.gaslimit** is this block's gas capacity.
- **block.number** is the current block height.
- **block.timestamp** is the block's real-world timestamp.
- **block.blockhash**(**uint** blockNumber) **returns**(**bytes32**) lets us fetch the latest 256 block hashes (exceling the current block hash).

Bonus question: Why can we not fetch the current block hash?

1.2 Global clock

Ethereum has two global clocks, **block.number** and **block.timestamp**, which lets a smart contract measure time. We must stress that a smart contract cannot act in *real-time* and it only has a rough estimate of current time.

Block numbers Ethereum has a single blockchain that grows linearly. Every block has a *height* and this acts as a monotonically increasing counter. While miners try to create blocks every 12 seconds, this is only an approximation due to the probabilistic nature of mining. Thus it is difficult to predict exactly when a block at height t will be minted.

Timestamp Ethereum has two consensus rules for a block's timestamp. It must be strictly greater than the previous block, and less than 900 seconds into the future from a peer's local clock.[1] Thus a peer will only a block if its timestamp is strictly increasing and it is close to real-world time. As a side note, the timestamp guarantee in Bitcoin is *even weaker*. A newly minted block's timestamp must be within 2 hours of a peer's network-adjusted time[2] and be greater than the median timestamp of the previous 11 blocks. This lets a block's timestamp be greater or less than the previous block, and significant flexibility in setting a fake timestamps.

Miner interference While miners cannot interfere with the block height, they can set the block's timestamp. This power does not come for free though as a block's timestamp is used to decide the difficulty target (i.e. how difficult will the puzzle become?). For example, if miners increment every block's timestamp by one second, then the mining difficulty will sky-rocket and miners will be forced to abandon the fork. On the other hand, if a block's timestamp is too far in the future, it'll be rejected by peers on the network. [3]

1.3 Basic Cryptographic Operations

The EVM supports cryptographic hash functions and verifying digital signatures.

Hash Functions A code example is provided in Figure 1. In cryptography, a hash function can be used to build a *commitment scheme* that lets a party commit to a chosen value (or statement) while keeping it hidden to others, with the ability to reveal the committed data later. It is a one-way function that takes as input a *pre-image* (i.e. arbitrary sized data) and computes a *hash* (a pseudo-random, but deterministic fixed output).

Ethereum natively supports **sha256** and **keccak256** (the pre-competition version). We must encode one or more variables into **bytes** before it can be used by the hash function. One way is to use **abi.encodePacked**(item1,...,itemN) which packs multiple variables.

[1] Taken from https://github.com/ethereum/wiki/blob/c02254611f218f43cbb07517ca8e5d00fd6d6d75/Block-Protocol-2.0.md

[2] This time depends on which peers it connects too

[3] A fun side-project may involve simulating the success of miner's interfering with a block's timestamp.

Hash Function

```
function hashSingle(bytes32 item) public pure returns(bytes32) {
  return sha256(abi.encodePacked(item));
}

function hashDouble(bytes32 item1, bytes32 item2) public pure
                                                    returns(bytes32) {
  return keccak256(abi.encodePacked(item1,item2));
}
```

Fig. 1: How to hash one or more items

Verify Signatures Ethereum (and other cryptocurrencies) use the Elliptic Curve Digital Signature Algorithm (ECDSA). Each party has a private key d and this is used to compute their public key $P = dG$, where G is a well-known generator. Digital signatures have a signing and verification function:

$$(r, s) = \texttt{Sign}(d, H(msg))$$
$$b = \texttt{Verify}(P, H(msg), (r, s))$$

A user signs the hash of a message, and not the actual message. To sign, it requires the message hash and the signer's private key d. In the signature (r, s), r is a random nonce and s represents the signed component. Verifying the signature (r, s) requires the message hash and the signer's public key. It returns b which is either 1 (true) or 0 (false). For efficiency, the EVM does not perform the verification algorithm. Instead, it natively supports re-computing the signer's Ethereum account P' given a signature (r, s) and the message hash h.

P' = **ecrecover**(**bytes32** hash, **uint8** v[4], **bytes32** r, **bytes32** s).

Thus, we recover the signer's public key from the signature and if $P == P'$.

1.4 Preconditions and Modifiers

A smart contract is a global program and it can be executed by anyone in the world. It is the developer's duty to ensure a function can only be executed when it *makes sense* and by the *appropriate users*. Abstractly, we can say that a *list of preconditions* must be satisified before a function can be executed.

Let's consider a simple withdrawal function. Only players with a deposit can invoke `withdraw()` (condition 1), and it can only be called when the game is in a 'finished' state (condition 2). We'll cover how to incorporate preconditions in a smart contract before introducing modifiers which let us append a list of pre-conditions to a new function.

[4] Not important for our tutorial. It lets us re-compute the y co-ordinate for r.

Preconditions In Figure 2, we provide a simple example of how to use both require and assert. A precondition is essentially *a conditon that must be fulfilled before other things can happen or be done*. We'll consider two keywords in Solidity to let us check that a precondition is satisified:

- **require**() is typically used at the start of a function call to check it inputs. If the precondition fails, the remaining gas is refunded and all execution so far is reverted.
- **assert**() is typically used to validate the new state after execution. If the assert fails, all remaining gas is used up and all execution so far is reverted.

Note the crucial technical difference. **require** refunds the remaining gas, whereas **assert** will burn the caller's remaining gas.

Require and Assert.

```
function withdraw(uint256 toWithdraw) public payable {
  require(balance[msg.sender] >= toWithdraw);
  // Update state to deduct the balance of msg.sender
  // Send coins to msg.sender
  assert(balance[msg.sender] >= 0); // Check for underflow
}
```

Fig. 2: Simple code example of require and assert.

Modifiers We provide a code sample in Figure 3. A **modifier** lets the developer attach code that is always executed before the desired function. It is typically used to attach precondition checks that are common across two or more functions. There are two key points to remember:

- Functions can support one or more modifiers, but every modifier must be listed as part of a function's definition.
- A modifier's code will always be executed first and the smart contract will only continue to execute a function if the marker _; is reached.

The best known example is onlyOwner() which only lets a contract's owner execute a function.

onlyOwner modifier

```
modifier onlyOwner() {
  require(msg.sender == owner);
  _; // The underscore lets us continue to the function's execution
}

function updateUsers(address[] users) public pure onlyOwner {
  // Code to update list of users
}
```

Fig. 3: How to restrict function calls to the contract's owner.

2 Exercise: Handling Money

There are two fundamental features of a cryptocurrency.

- Global synchronisation of data
- Self-enforced and global value transfer

The blockchain is essentially a cryptographic audit log that helps us ensure every computer in the world can be synchronised and compute the same global ledger of accounts. In this tutorial we'll focus on the value-transfer feature and we'll explore how a smart contract can self-enforce it.

2.1 Deposit and Refund Paradigm

A principle of cryptocurrencies is that every party has a *financial motivation* to participate. This brings us to the *deposit and refund paradigm* which lets a smart contract withhold a party's deposit until they have finished the protocol. In a way, it can be used to 'nudge' parties towards honest participation (and not abort during a protocol's execution). Let's consider a concrete example.

Seal-bid Auctions Let's consider a group of parties who want to bid for a digital service. There are two stages in this auction smart contract:

- All parties submit a sealed bid before time t_1.
- All parties must open their bid between t_1 and t_2.

We want to ensure that every sealed bid is opened during the auction without the help of an external (and trusted) auctioneer. A smart contract can require a deposit to be associated with each sealed bid, and the deposit is simply lost if the bid isn't opened before t_2. This provides a financial motivation for every bidder to finish the protocol (and open their bid) to get their deposit back, even if they haven't won.

Deposit Function Let's create a new smart contract (*DepositAndRefund.sol*) for this exercise. We'll first focus on creating the necessary data structures, and how to keep a record of a party's full deposit.

- Set up a **mapping** that links an **address** to a **uint**.
- Create the function `getBalance(address party)` to fetch a party's balance.
- Create the function `deposit()` to receive and record a party's deposit. (hint: check the sender and value using **msg**, and don't forget the **payable** keyword).
- Try sending coins to the contract in Remix using your new deposit function (and don't forget to check your balance too).

Checks-effects-interactions As we'll learn in greater detail next week, a pseudonymous attacker can drain every coin from our smart contract if the withdraw function is not designed well. This brings us to the `checks-effects-interactions` heuristic that was proposed in response to TheDAO hack:

- Perform all pre-condition *checks* to ensure a function should be executed,
- Perform all *effects* by updatin the contract's state,
- Finally *interact* with other contracts and accounts.

The withdrawal function is a great example for the above heuristic. First, we should check **msg.sender** has a sufficient balance to withdraw. Second, we should deduct **msg.sender**'s balance by the number of coins they wish to withdraw. Third, our smart contract can send **msg.sender** their coins.

Withdrawal function With the above in mind, it is time for write the withdaw function:

- Define the `withdraw(uint amount)` function (Hint: don't forget **payable**).
- Include a precondition check to confirm the caller's balance covers the withdrawal request.
- Update the caller's balance by deducting the number of coins they are withdrawing.
- Transfer the caller their coins (Hint: look at last week's tutorial sheet).
- Try sending and withdrawing coins from the contract in Remix.

And that's it! You now have a contract that can support basic deposit and withdrawal functionality.

2.2 Extra 'bonus' tasks

Let's test our understanding We've covered a lot of content in the past two tutorials. Let's take this opportunity to dive a bit deeper into the differences between 'similar' key words.

- What is the difference between **assert** and **require**?

- What is the difference between **byte**, **byte1** and **bytes**?
- What combinations of key words lets us track the total gas used within a function?
- What is the difference between **tx.origin** and **msg.sender**? And why is it not recommended practice to use **tx.origin**?
- What is the difference between **address.transfer** and **address.send**? And why do both functions only forward 2,300 gas to the receiver?

Timed Withdrawal Some smart contracts will only let parties withdraw their coins after time t. Try to modify your new smart contract to kick-start a timer after a party has deposited coins and only let them withdraw their coins after the timer has expired. Hint: We discussed in Section 1.2 how time is handled in Solidity. As well, you may want to store a seperate timer for each party.

Verify a digital signature? In Section 1.3, we presented how to verify a digital signature in Solidity using **ecrecover**. Try updating the withdraw() function to only send a party their deposit if they sign the phrase 'to the moon'. This will take time as you wll also need to prepare a signed message too (and you will also need to consider if **msg.sender** is required anymore).

ERC20 Tokens We have mostly focused on how to handle payments in ether, but over the past few years we have witnessed the rise of ERC20 tokens that let parties mint/issue their own coins. Take this opportunity to find information online about ERC20 tokens, and try to answer the following questions:

- Where can we find the balance of each party for a given token?
- What does it mean to *transfer* an ERC20 token to another party? And roughly how do we do that?
- What are the necessary steps to deploy your own ERC20 token?