# 1 Basic Serpent Contract Example

Before moving into more difficult examples, let's take a quick look at an Easy Bank example from KenK's first tutorial. A contract like this allows for a fully transparent bank to function with an open ledger that can be audited by any node on the network (an ideal feature for ensuring banks aren't laundering money or lending to enemies of the state.)

Before looking at the code for the contract, let's define our "easy bank" further. Our bank will be using its own contracual currency and not Ether (which we will dicuss and implement in a later contract example). So creating the currency is done within our contract. Now that we know what our bank does (create and send a currency that is exclusive to the contract), let's define what the contract must be capable of doing:

1. Setup at least one account with an initial balance of our contract-exclusive currency
2. Take funds from one account and send our currency to another account

```
def init():
        self.storage[msg.sender] = 10000
        self.storage["Taylor"] = 0
def send_currency_to(value):
        to = "Taylor"
        from = msg.sender
        amount = value
        if self.storage[from] >= amount:
                self.storage[from] = self.storage[from]  - amount
                self.storage[to] = self.storage[to] + amount
```

So what's going on in this contract? Our contract is divided into two methods, let's take a look at the first method:

```
def init():
        self.storage[msg.sender] = 10000
        self.storage["Taylor"] = 0
```

The init function in serpent is very similar to init in python, which is very similar to common constructors in Java. The init function runs once and only once at contract creation. In our contract the init block runs and instantiates two objects in contract storage.

Our init method, from a general perspective, intializes one account with a balance of 10,000U (this will be how we donote our contract-exclusive currency) and another account with a balance of 0U. In our Ethereum contract, storage is handled with key value pairs. Every contract has their own storage which is accessed by calling self.storage[key]. So in our example the easy bank's contract storage now has a value of 10,000U at key msg.sender (we'll identify what this is in a moment) and at the key "Taylor" there is a value of 0U.

Awesome. So who is msg.sender? msg.sender is the person who is sending the specific message to the contract - which in this case is us. msg.sender is unique and assigned and verified by the network. We now have a heightened understanding of init, lets look at our send method.

```
def send_currency_to(value):
```

```
to = "Taylor"
from = msg.sender
amount = value
if self.storage[from] >= amount:
        self.storage[from] = self.storage[from] - amount
        self.storage[to] = self.storage[to] + amount
```

Let's take a look at this one piece at a time. The first three lines aim are setting up variables that we will use in the last three lines. The first line is establishing who we are sending our funds to - and just as we setup in init, we are sending our funds to our friend Taylor. from is being set to the address that the funds are from, which is us - msg.sender. Finally, the value variable is set to the parameter passed to our send_currency_to function. When the contract was invoked a value needed to be sent in order for it to run properly, this value is the value that is going to be sent to Taylor.

Okay, now that we understand what variables we are working with let's dive into the last portion of our contract. We want to check that the balance for the bank account in the contract's storage at from (remember that you are who this is from!) is greater than or equal to the amount we are attempting to send - obviously we do not want our contract sending money that the sender does not have.

If the account balance passes our check we subtract the amount being sent from the sender balance:self.storage[from] = self.storage[from] - value. We then add to the balance of the account receiving the currency:self.storage[to] = self.storage[to] + value.

Great! We have officially worked our way through a very basic contract example! Now our friend Taylor has 1000U! Try to think of ways that you could improve this contract, here are some things to consider:

- What happens when the value exceeds the amount setup in the 'from' account?
- What happens when the value is negative?
- What happens when value isn't a number?

## 2 Moderate Serpent Contract Example

So we've made it through the first serpent example, which we now have realized wasn't as daunting as it first seemed. We understand that every contract has its own contractual storage that is accessed through self.storage[key] = value. We understand that we can use parameters passed to function. Lastly we understand that msg.sender gives us the unique identifier of whoever sent the message, and that all participants involved with a contract have their own unique identifier that can be used in whatever creative way you like.

Let's look at a more moderate contract that keeps with our bank theme. So, just like with our first contract, we need to classify what we are making and what characteristics the contract will need to leverage our desired features. We are going to be implementing what is known as a mutual credit system. A generalized idea of a mutual credit system is the intersection of a barter system and a non-regulated currency model. So, let's define a community that implements a mutual credit system and every participant gets a 1000 Unit credit (in this case 1UC = 1USD). In the beginning

there is no money at all. It only comes into circulation when one of the participants uses his credit to pay another participant. If he uses his 1000 Units his balance is -1000 U. His supplier's balance is now +1000U. The total amount in circulation is now also 1000 U. This means there is always exactly as much in circulation as there is outstanding credit: a zero sum game.

One can clearly notice that this system creates money at the time of the transaction. At time 0, before any of the community's participants completed a transaction, the currency in circulation was zero. It is also clear that, unlike fiat currencies, this model does not require any centralized money supply management which, when discussing decentralized apps built on blockchain technology, is an attractive idea to implement. Regardless of your opinion on such a system, let's automate a contract to initiate these transactions and act as a public ledger to keep track of the community's participants and their account balances.

```
def init():
        contract.storage[((msg.sender * 0x10) + 0x1)] = 0x1
        contract.storage[((msg.sender * 0x10) + 0x2)] = 0x1

def code():
        toAsset = (msg.data[0] * 0x10) + 0x1
        toDebt = (msg.data[0] * 0x10) + 0x2
        fromAsset = (msg.sender * 0x10) + 0x1
        fromDebt = (msg.sender * 0x10) + 0x2
        value = msg.data[1]

        if contract.storage[fromAsset] >= value:
                contract.storage[fromAsset] = contract.storage[fromAsset] - value
        else:
                contract.storage[fromDebt] = value - contract.storage[fromAsset]
                contract.storage[fromAsset] = 0

        if contract.storage[toDebt] >= value:
                contract.storage[toDebt] = contract.storage[toDebt] - value
        else:
                value = value - contract.storage[toDebt]
                contract.storage[toAsset] = contract.storage[toAsset] + value
                contract.storage[toDebt] = 0
```

## 3 An Advanced Contract Example

Now that we have gone through and annotated several contract examples it is time to consider a couple key design concepts required to create a high-level smart contract. By the end of this section we will talk about several key mistakes that show up in high-level contracts, and you will aim to

identify and resolve them in a rock, paper, scissor contract example (RPS).

### 3.1 Contract Theft

The first contract design error we will talk about is contracts causing money to disappear. Some contracts require the participants to send an amount of money to enter the contract (lotteries, games, investment apps). All contracts that require some amount of money to participate have the potential to have that money lost in the contract if things don't go accordingly. Below is the add_player function from our RPS contract. The function adds a player and stores their unique identifier (msg.sender). The contract also takes a value (msg.value) that is sent to the contract. The value is the currency used by ethereum, Ether. Ether can be thought of in a similar light to Bitcoin; Ether is mined and used as the currency to fuel all contracts as well as the currency that individuals will trade within contracts. Let's dive in and see if we can find a contract theft error below:

```
def add_player():
        if not self.storage["player1"]:
                self.storage["player1"] = msg.sender
                self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                return(1)
        elif not self.storage["player2"]:
                self.storage["player2"] = msg.sender
                self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                return(2)
        else:
                return(0)
```

In this section, a user adds themselves to the game by sending a small amount of Ether with the transaction. The contract takes this Ether, stored in msg.value, and adds it to the winnings pool, the prize that the winner of each round will receive. Let's consider two very probable scenarios 1) a potential entrant sends too much Ether or too little Ether or, 2) the entrant is not allowed in the game for any reason. In both of the following scenarios the contract will keep their money. The first scenario allows an unfair entrance fee, where one user could bet below the expected amount and the other user pays the expected amount - both participants should be betting the same amount! The second scenario results in our contract taking Ether from those who aren't necessarily the two locked-in participants. Both of these errors will cause distrust in our contract, eventually resulting in the community not trusting this contract and its author - you.

It seems like our contract could use some case checks on when to issue refunds - think about how you would do this. Go ahead and try it and see if your idea works! Are there any other edge cases where issuing a refund should be considered?

### 3.2 Failing to use Cryptography

It goes without saying that as a student in a computer security course you would implement cryptographic practices wherever you can. Thus with any contract that requires any user inputs that affect the outcome of said contract, crypto should be implemented. In our RPS contract the user is using a numeric scale as their input with 0: rock, 1: paper, 2: scissors. Let's take a look at the function that registers their inputs.

```
def do(a):
        if self.storage["player1"] == msg.sender:
                self.storage["p1value"] = a
                return(1)
        elif self.storage["player2"] ==  msg.sender:
                self.storage["p2value"] = a
                return(2)
        else:
                return(0)
```

We can see that our do() function identifies the sender with msg.sender and then stores their input 'a' in plaintext (where a = 0, 1, or 2). The lack of encryption means that the other player could see what their opponent played by looking at a block that published it; with that information they could input their choice to ensure they always win. This can be fixed by hashing (with a nonce) the value to send to the contract. Then, when both parties have decided what to play, they send their answer and a nonce to show what they played. Understanding where crypto elements should be used is crucial to justifying your contract.

In order to enhance the security and fairness of our contract we will use the hashing functions discusses here[INSERT LINK TO HASING SECTION HERE]. The first change that is necessary in our contract is to have the do() function accept the hash given from the user. Our RPS application would prompt the participants in our game to send a hash of their input and a nonce of their choosing. Thus a = SHA256(numerical input (0 or 1 or 2) + nonce). This hashed value is stored in the contract, but there is no way for either opponent to discover the other's input.

Now that we have the hash stored in the contract we need to look at the check() function:

```
def check():
        if

        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        else:
                return(-1)
```

In its current state this function will no longer work because 'p1value' and 'p2value' no longer have the cleartext input, they have a randomized hash. Thus, we need will modify the check() so the user will send their plaintext input and their nonce as parameters so check() can verify that what they gave matches the hash that is stored in the contract. Remember, up until this point the contract has *no way of knowing* who the winner is because it has *no way of knowing* what the inputs are. The contract doesn't know the nonce, so it cannot understand what the input a send to do() was. Now that we know that check() was verify the answer, below is the updated, cleaned up contract:

```
def do(player_hash):
        if self.storage["player1"] == msg.sender:
                self.storage["p1hash"] = player_hash
                return(1)
        elif self.storage["player2"] ==  msg.sender:
                self.storage["p2hash"] = player_hash
                return(2)
        else:
                return(0)

def verify(a, nonce):
        if self.storage["player1"] == msg.sender:
                player_one_hash = array(1)
                player_one_hash[0] = a + nonce
                self.storage["p1value"] = a
        elif self.storage["player2"] == msg.sender:
                player_two_hash = array(1)
                player_two_hash[0] = a + nonce
                self.storage["p2value"] = a
        else:
                return(-1)

def check():
        if self.storage["p1value"] and self.storage["p2value"]:
                if self.storage["p1value"] == self.storage["p2value"]:
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                return(0)
                elif self.storage["p1value"] > self.storage["p2value"] and self.sto
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else
```

```
                        pay_out_to("player1")
                        return(1)
                elif self.storage["p1value"] > self.storage["p2value"] and self.sto
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player2")
                                return(2)
                elif self.storage["p2value"] > self.storage["p1value"] and self.sto
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player2")
                                return(2)
                elif self.storage["p2value"] > self.storage["p1value"] and self.sto
                        if verify("player1") == 0:
                                pay_out_to("player2")
                        elif verify("player2") == 0:
                                pay_out_to("player1")
                        else:
                                pay_out_to("player1")
                                return(1)
        else:
                return(-1)

def pay_out_to(player):
        send(self.storage[player], self.storage["WINNINGS"])
```

### 3.3 Incentive incompatability

The final key bug to watch out for is incentive incompatibility. There are contract ideas that must consider user incentives in order for them to run as planned. If I had an escrow contract incentives must be implemented so both individuals don't always so they did not receive their promised service. If I have a game contract where inputs are encrypted, incentives must be implemented to ensure both players decrypt their responses simultaneously to avoid cheating. Let's look and see how our RPS contract holds up with regard to incentives:

```
def check():
        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p
                send(self.storage["player1"], self.storage["WINNINGS"])
```

```
            return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p
            send(self.storage["player2"], self.storage["WINNINGS"])
            return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p
            send(self.storage["player2"], self.storage["WINNINGS"])
            return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p
            send(self.storage["player1"], self.storage["WINNINGS"])
            return(1)
        else:
            return(-1)
```

Given the version at the end of this section our contract is incentive compatible. Only one party needs to call the check() function in order for the winnings to be fairly distributed to the actual winner, regardless of who calls. If we implemented the crypto elements from part 2 of this section we would need to wait for a certain number of blocks to be mined and published before checking the results. If someone didnt bother to decrypt their rock/paper/scissors within that timeframe, the contract would, by default, send the money to the person who *did* decrypt their input. This incentivizes both users to decrypt their inputs before the check() function is called after a random amount of blocks have been published; if you don't decrypt you are *always* guaranteed not to win.

### 3.4 Rock, Paper, Scissor Contract

```
def init():
        self.storage["MAX_PLAYERS"] = 2
        self.storage["WINNINGS"] = 0

def add_player():
        if not self.storage["player1"]:
            self.storage["player1"] = msg.sender
            self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
            return(1)
        elif not self.storage["player2"]:
            self.storage["player2"] = msg.sender
            self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
            return(2)
        else:
            return(0)

def do(a):
        if self.storage["player1"] == msg.sender:
            self.storage["p1value"] = a
            return(1)
        elif self.storage["player2"] ==  msg.sender:
            self.storage["p2value"] = a
```

```
                return(2)
        else:
                return(0)

def check():
        if self.storage["p1value"] == self.storage["p2value"]:
                return(0)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p1
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        elif self.storage["p1value"] > self.storage["p2value"] and self.storage["p1
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p2
                send(self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        elif self.storage["p2value"] > self.storage["p1value"] and self.storage["p2
                send(self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        else:
                return(-1)

def balance_check():
        log(self.storage["player1"].balance)
        log(self.storage["player2"].balance)
```