A Programmer's Guide to Ethereum and Serpent

Kevin Delmolino
`del@terpmail.umd.edu`

Mitchell Arnett
`marnett@umd.edu`

Ahmed Kosba
`akosba@cs.umd.edu`

Andrew Miller
`amiller@cs.umd.edu`

Elaine Shi
`elaine@cs.umd.edu`

April 23, 2015

Contents

1 Introduction

The goal of this document is to teach you everything you need to know about Ethereum in order to start developing your own Ethereum contracts and decentralized apps. So, what is Ethereum? Ethereum can be seen as a decentralized platform that uses the network unit Ether as the fuel to power all contracts on the network. Ethereum is more than a cryptocurrency (even though mining is involved), it is a network that enables and powers Ethereum contracts. So what is an Ethereum contract? Think of it as a program that aims to provide decentralized services including: voting systems, domain name registries, financial exchanges, crowdfunding platforms, company governance, self-enforcing contracts and agreements, intellectual property, smart property, and distributed autonomous organizations. Ethereum is the ubiquitous bitcoin. It uses a similar underlying blockchain technology as bitcoin while broadening the scope of what it is capable of accomplishing. [**?**, **?**]

2 Ethereum Tools

2.1 Acquiring the Virtual Machine

We have made a virtual machine that contains all of the necessary software. The virtual machine is running Ubuntu 14.04 LTS, Pyethereum and Serpent 2.0. Pyethereum is the program that allows for us to interact with the blockchain and test our contracts. We will be using Pyethereum, but there are also Ethereum implementations in C++ (cpp-ethereum) and Go (go-ethereum). Serpent 2.0 will allow for us to compile our serpent code into the stack-based language that is actually executed on the blockchain.

The virtual machine requires the host to be a 64-bit operating system, and for optimal performance, hardware acceleration should be turned on (VT-d/AMD-V). Normally, this is turned on by default when supported by your processor. Due to the advanced graphics used on the Ubuntu desktop, we recommend turning on 3D acceleration. For more information, refer to your virtual machine's documentation.

The Virtual Machine has been tested using VMWare Fusion (`https://www.vmware.com/products/fusion`) and VirtualBox (`https://www.virtualbox.org/`), however, it should work with any VM software that supports VMDK files. The Virtual Machine is available from `https://drive.google.com/file/d/0BzlG8wGYwTrGWlp0LWctYVIxRVU/view?usp=sharing`. The username is "user" and the password is "dees".

2.2 Installing Pyethereum and Serpent

NOTE: This section is not required if the provided virtual machine is used. We have preinstalled all of the necessary applications to program Ethereum contracts using Pyethereum and Serpent. This section goes over installing a native copy of Pyethereum and Serpent on your machine and give a brief overview of what each component does.

This section assumes you are comfortable with the command line and have git installed. If you need assistance getting git installed on your local machine, please consult `http://git-scm.com/book/en/v2/Getting-Started-Installing-Git`.

First, lets install Pyethereum. In order to install Pyethereum, we first need to download it. Go to a directory you don't mind files being downloaded into, and run the following command:

```
git clone https://github.com/ethereum/pyethereum
```

This command clones the code currently in the ethereum repository and copies it to your computer. Next, change into the newly downloaded pyethereum directory and execute the following command

```
git branch develop
```

This will change us into the develop branch. This code is usually stable, and we found that it has better compatibility with the more modern versions of Serpent. Please note that later on, this step may not be necessary as the Ethereum codebase becomes more stable, but with the current rapid development of Ethereum, things are breaking constantly, so it pays to be on the cutting edge.

Finally, we need to install Pyethereum. Run the following command:

```
python setup.py install --user
```

This actually installs Pyethereum on our computer. Note that commands may be different if you are on a non-Unix-like platform. We recommend running Ethereum on Unix-like operating systems such as Mac OS X and Linux.

Now, we are going to install serpent. The steps are extremely similar. Go to the directory that you downloaded ethereum into and run the following commands:

```
git clone https://github.com/ethereum/serpent
cd serpent
git branch develop
python setup.py install --user
```

Now that Pyethereum and Serpent are installed, we should test that they are working. Go to the pyethereum/tests directory and run the following command:

```
python pytest -m test_contracts.py
```

If the test states that it was successful, then everything is installed correctly and you are ready to continue with this guide!

3 Using Pyethereum Tester

In order to test our smart contacts, we will be using the Pyethereum Tester. This tool allows for us to test our smart contracts without interacting with the blockchain itself. If we were to test on a real blockchain - even a private one - it would take a lot of time to mine enough blocks to get our contract published on the blockchain and to run commands on it. Therefore, we use the tester.

Below is a simple contract that will be used as an example to show how to set up a contract. [?, ?]

```
import serpent
from pyethereum import tester, utils, abi

serpent_code='''
def main(a):
        return (a*2)
'''

evm_code = serpent.compile(serpent_code)
translator = abi.ContractTranslator(
        serpent.mk_full_signature(serpent_code))
data = translator.encode('main', [2])
s = tester.state()
c = s.evm(evm_code)
o = translator.decode('main', s.send(tester.k0, c, 0, data))
```

```
print(o)
```

Now what is this code actually doing? Let's break it down.

```
import serpent
from pyethereum import tester, utils, abi
```

This code imports all of the assets we need to run the tester. We need *serpent* to compile our contract, we need pyethereum *tester* to run the tests, we need *abi* to encode and decode the transactions that are put on the blockchain, and we need *utils* for a few minor operations (such as generating public addresses).

```
serpent_code='''
def main(a):
        return (a*2)
'''
```

This is our actual serpent code. We will discuss Serpent's syntax later in the guide, but this code will return a value that is double the parameter a. Please note that the code between the triple quotes is the only non-python code in this section.

```
evm_code = serpent.compile(serpent_code)
translator = abi.ContractTranslator(
        serpent.mk_full_signature(serpent_code))
```

Here, we finally get ready to run our actual code. The *evm_code* variable holds our compiled code. This is the byte code that we will actually "run" using ethereum. The translator variable holds the code that will allow for us to encode and decode the code that will be run on the blockchain.

```
data = translator.encode('main', [2])
s = tester.state()
```

The data variable holds our encoded variables. We are going to call the *main*() function, and we are going to send one parameter to it, the number 2. We encode using the translator. Next, we are going to create a state (essentially a fake blockchain). This state is what we will run our contract on.

```
c = s.evm(evm_code)
o = translator.decode('main', s.send(tester.k0, c, 0, data))
```

The c variable holds our contract. The *evm*() function puts our contract onto our fake blockchain. Finally, we run a transaction. We use the *send*() function to execute the transaction on the contract (whose address is stored in c). The entity sending the transaction is $tester.k0$ who is a fake private key used for testing. It signs and "authorizes" the transaction. We are sending no ether into the contract, so the third parameter is a zero. Finally, we send our encoded data.

```
o = translator.decode('main', s.send(tester.k0, c, 0, data))
print(o)
```

Finally here, we will use our translator to decode out what the function returned. We will print that using the standard python print() function.

The code can be executed using the command "python file_name.py". When executed, this code will output double the input parameter. So this code will output the number 4. [?, ?]

3.1 Public and Private Keys

All cryptocurrencies are based on some form of public key encryption. What does this mean? It means that messages can be encrypted with one key (the private key) and unencrypted with the public key. The Pyethereum tester provides us with fake addresses we can use for testing (tester.k0 - tester.k9), each of them representing an individual party in the contract. However, these are private addresses that we are using to sign transactions. This tells the world that we have authorized this transaction to exist. Others can confirm this by using our public key.

Now, lets say we want someone to be able to submit public keys to a contract as a parameter. How do we calculate the public keys from the private tester keys we have? There is a function in pyethereum's utils that allows for us to do this:

```
public_k1 = utils.privtoaddr(tester.k1)
data = translator.encode('transfer', [500, public_k1])
```

We don't want to send our private key to a contract, because then others could sign transactions as us and take all of our ether! The code above uses the $utils.privtoaddr(private_key)$ function, which returns the public key associated with $private_key$. We can then send the public key with the transaction, as we do in line two.

4 Language Reference

There are several different languages used to program smart contracts for Ethereum. If you are familiar with C or Java, Solidity is the most similar language. If you really like Lisp or functional languages, LLL may be the language for you. The Mutant language is most similar to C. We will be using Serpent 2.0 (we will just refer to this as Serpent, since Serpent 1.0 is deprecated) in this reference, which is designed to be very similar to Python. Even if you are not very familiar with Python, Serpent is very easy to pickup. Note that all code after this point is Serpent, not Python. In order to test it, it must be put in the $serpent_code$ variable mentioned previously. Another thing to note is that many, if not all, of the built-in fuctions you may come accross in other documentation for Serpent 1.0 will work in 2.0. I

4.1 The log() Function

The *log*() function allows for easy debugging. If X is defined as the variable you want output, $log(X)$ will output the contents of the variable. We will use this function several times throughout this document. Here is an example of it in use:

```
def main(a):
        log(a)
        return(a)
```

This code will output the variable stored in a. Since we passed in a three, it should be a three. Below is the output of the log function:

```
('LOG', 'c305c901078781c232a2a521c2af7980f8385ee9', [3L], [])
```

The part that is important to us is the third piece of data stored in the tupple, specifically, the $[3L]$. This tells us that the value in the variable is a three.

4.2 Variables

Assigning variables in Serpent is very easy. Simply set the variable equal to whatever you would like the variable to equal. Here's a few examples:

```
a = 5
b = 10
c = 7
a = b
```

If we printed out the variables a, b and c, we would see 10, 10 and 7, respectively.

Special Variables Serpent creates several special variables that reference certain pieces of data or pieces of the blockchain that may be important for your code. We have reproduced the table from the official Serpent 2.0 wiki tutorial (and reworded portions) for your reference below. [**?**]

Variable	Usage
tx.origin	Stores the address of the address the transaction was sent from.
tx.gasprice	Stores the cost in gas of the current transaction.
tx.gas	Stores the gas remaining in this transaction.
msg.sender	Stores the address of the person sending the information being processed to the contract
msg.value	Stores the amount of ether (measured in wei) that was sent with the message
self	The address of the current contract
self.balance	The current amount of ether that the contract controls
x.balance	Where x is any address. The amount of ether that address holds
block.coinbase	Stores the address of the miner
block.timestamp	Stores the timestamp of the current block
block.prevhash	Stores the hash of the previous block on the blockchain
block.difficulty	Stores the difficulty of the current block
block.number	Stores the numeric identifier of the current block
block.gaslimit	Stores the gas limit of the current block

Wei is the smallest unit of ether (the currency used in ethereum). Any time ether is referenced in a contract, it is in terms of wei. There are several other denominations as seen in the table below [**?**]:

Denomination	Amount (in ether)
wei	1.0×10^{18}
Kwei	1.0×10^{15}
Mwei	1.0×10^{12}
Gwei	1.0×10^{9}
Szabo	1.0×10^{6}
Finney	1000
Ether	1
Kether	.001
Mether	1.0×10^{-6}
Gether	1.0×10^{-9}
Tether	1.0×10^{-12}

A very easy to use converter is available at `http://ether.fund/tool/converter`.

4.3 Control Flow

In Serpent, we mostly will use if..elif..else statements to control our programs. For example:

```
if a == b:
        a = a + 5
        b = b - 5
        c = 0
        return(c)
elif a == c:
        c = 5
        return(c)
else:
        return(c)
```

Tabs are extremely important in Serpent. Anything that is inline with the tabbed section after the if statement will be run if that statement evaluates to true. Same with the elif and else statements. This will also apply to functions and loops when we define those later on. [?]

Important to also note is the *not* modifier. For example, in the following code:

```
if not a == b:
        return(c)
```

The code in the if statement will not be run if a is equal to b. It will only run if they are different. The *not* modifier is very similar to the ! modifier in Java and most other languages. [?]

4.4 Loops

Serpent supports while loops, which are used like so:

```
somenum = 10
while somenum > 1:
        log(somenum)
        somenum = somenum - 1
```

This code will log each number starting at 10, decrementing and outputting until it gets to 1. [?]

4.5 Arrays

Arrays are very simple in serpent. A simple example is below:

```
def main():
        arr1 = array(1024)
        arr1[0] = 10
        arr1[129] = 40
        return(arr1[129])
```

This code above simply creates an array of size 1024, assigns 10 to the zero-th index and assigns 40 to index 129. It then returns the value at index 129 in the array [?, ?].

Functions that can be used with Arrays include:

- slice(arr, items=s, items=e) where arr is an array, s is the start address and e is the end address. This function splits out the portion of the array between s and e, where $s <= e$. That portion of the array is returned.
- len(arr) returns the length of the arr array.

Returning arrays is also possible [?]. In order to return an array, append : arr to the end of the array in the return statement. For example:

```
def main():
        arr1 = array(10)
        arr1[0] = 10
        arr1[5] = 40
        return(arr1:arr)
```

This will return an array where the values were initialized to zero and address 0 and 5 will be initialized to 10 and 40, respectively [?].

4.6 Strings

Serpent uses two different types of strings. The first is called short strings. These are treated like a number by Serpent and can be manipulated as such. Long strings are treated like an array by serpent, and are treated as such. Long strings are very similar to strings in C, for example. As a contract programmer, we must make sure we know which variables are short strings and which variables are long strings, since we will need to treat these differently. [?]

Short Strings Short strings are very easy to work with since they are just treated as numbers. Let's declare a couple new short strings:

```
str1 = "string"
str2 = "string"
str3 = "string3"
```

Very simple to do. Comparing two short strings is also really easy:

```
return (str1 == str2)
return (str1 == str3)
```

The first return statement will output 1 which symbolizes true while the second statement will output 0 which symbolizes false. [?]

Long Strings Long strings are implemented similarly to how they are in C, where the string is just an array of characters. There are several commands that are used to work with long strings:

- In order to define a new long string, do the following:

  ```
  arbitrary_string = text("This is my string")
  ```

- If you would like to change a specific character of the string, do the following:

  ```
  arbitrary_string = text("This is my string")
  setch(arbitrary_string, 5, "Y")
  ```

  In the setch() function, we are changing the fifth index of the string *arbitrary_string* to $'Y'$.

- If you would like to have the ASCII value of a certain index returned, do the following:

  ```
  arbitrary_string = text("This is my string")
  getch(arbitrary_string, 5)
  ```

  This will retrieve the ASCII value at the fifth index in *arbitrary_string*.

- All functions that work on arrays will also work on long strings.

[?, ?]

To check for the equality of two strings, it gets a little more difficult, and requires the *getch*() method. An example is given below that returns -1 if *str*1 and *str*2 are not equal, and 1 if they are.

```
def compare_equals():
        str1 = text("String 1")
        str2 = text("String 1")
        i = 0
        while i < len(str1):
                if getch(str1,i) != getch(str2,i):
                        return(-1)
                i = i + 1
        return(1)
```

4.7 Functions

Functions work in Ethereum very similarly to how they work in other languages. You can probably infer how they are used from some of the previous examples. Here is an example with no parameters:

```
def main():
        #Some operations
        return(0)
```

And here is an example with three parameters:

```
def main(a,b,c):
        #Some operations
        return(0)
```

Defining functions is very simple and makes code a lot easier to read and write [?]. But how do we call these functions from within a contract? We must call them using $self.function\_name(params)$. Any time we reference a function within the contract, we must call it from self (a reference to the current contract). Note that any function can be called directly by a user. For example, lets say we have a function A and a function B. If B has the logic that sends ether and A just does the check, and A calls B to send the ether, an aversary could simply call function B and get the ether without ever going through the checks. We can fix this by not putting that type of logic in separate functions.

Special Function Blocks There are three different special function blocks. These are used to declare functions that will always execute before certain other functions.

First, there is *init*. The *init* function will be run once when the contract is created. It is good for declaring variables before they are used in other functions.

Next, there is *shared*. The *shared* function is executed before *init* and any other functions.

Finally, there is the *any* function. The *any* function is executed before any other function except the *init* function [?].

4.8 Sending Wei

Contracts not only can have ether (currency) sent to them (via *msg.value*), but they can also send ether themselves. *msg.value* holds the amount of wei that was sent with the contract.

In order to send wei to another user, we use the send function. For example, lets say I wanted to send 50 wei to the user's address stored in x, I would use the code below.

```
send(x, 50)
```

This would then send 50 wei from this contract's pool of ether (the ether that other users/contracts have sent to it), to the address stored in x.

How do we get a user's address? The easiest way is to store it when that user sends a command to the contract. The user's address will be stored in *msg.sender*. If we save that address in persistent storage, we can access it later when needed [?] (we will go over persistent storage in the next section).

One thing to note is that the send function will send all of the remaining gas in the contract to the destination address, minus 25. if we want to define how much gas to send, we specify it as the first parameter. If we wanted to send only 100 gas, we would send the following:

```
send(100,x, 50)
```

4.9 Persistant Data Structures

Persistant data structures can be declared using the *data* declaration. This allows for the declaration of arrays and tupples. For example, the following code will declare a two dimensional array:

```
data twoDimArray[][]
```

Very simple, the next example will declare an array of tupples. The tupples contain two items each - $item1$ and $item2$.

```
data arrayWithTupples[](item1, item2)
```

These variables will be persistent throughout the contract's execution (In any command/-function called by any user to the same contract instance). Please note that data should not be declared inside of a function, rather should be at the top of the contract before any function definitions.

Now, lets say I wanted to access the data in these structures. How would I do that? Its simple, the arrays use standard array syntax and tupples can be accessed using a period and then the name of the value we want to access. Lets say, for example I wanted to access the $item1$ value from the $arrayWithTupples$ strucutre from the second array address, I would do that like so:

```
x = self.arrayWithTupples[2].item1
```

And that will put the $item1$ value stored in the $self.arrayWithTupples$ array into x. [?] Note that we will need the self declaration so the contract knows we are referencing the arrayWithTupples structure in this contract.

Self.storage[] Ethereum also supplies a persistent key-value store called $self.storage[]$. This is mostly used in older contracts and also is used in our example below for simplicity. Essentially, put the key in the brackets and set it equal to the value you want. An example is below when I set the value y to the key x.

```
self.storage["x"] = "y"
```

Now whenever $self.storage["x"]$ is called, it will return y. For simple storage, $self.storage[]$ is useful, but for larger contracts, we reccomend the use of data (unless you need a key value storage, of course). [?, ?]

4.10 Hashing

Serpent allows for hashing using three different hash functions - SHA3, SHA-256 and RIPEMD-160. The function takes the parameters a and s where a is the array of elements to be hashed and s is the size of the array to be hashed. For example, we are going to hash the array [4,5,5,11,1] using SHA-256 and return the value below. [?]

```
def main(a):
        bleh = array(5)
        bleh[0] = 4
        bleh[1] = 5
        bleh[2] = 5
        bleh[3] = 11
        bleh[4] = 1
        return(sha256(bleh, items=5))
```

The output is $[92958224028375895182299457531563411438064489993925166733548623543505998847 01L]$

The function definitions are:

- $x = sha3(a, size = s)$ for SHA3
- $x = sha256(a, size = s)$ for SHA-256
- $x = ripemd160(a, size = s)$ for RIPEMD-160

Please note that any inputs to the hash function can be seen by anyone looking at the block chain. Therefore, when keeping secrets between two parties, the hash values should be computed off of the blockchain then only the hash value put on the block chain. When we want to decode the secret in the hash, we should then send the nonce and the text to the blockchain, rehash it, and compare them with the prestored hash value. There is more detail about this process in the section ”Failing to Use Cryptography”.

4.11 Random Number Generation

In order to do random number generation, you must use one of the previous blocks as a seed. Then, use modulus to ensure that the random number is in the necessary range. In the following examples, we will do just this.

In this example, we will the function will take a parameter a. It will generate a number between 0 and a (including zero).

```
def main(a):
        raw = block.prevhash
        if raw < 0:
                raw = 0 - raw
        return(raw%a)
```

Note that we must make sure that the raw number is positive. [?]

If we wanted the lowest number to be a number other than zero, we must add that number to the random number generated.

Now, when we are referencing previous blocks, we need to make sure there are blocks before our current block that we can reference. On the actual ethereum blockchain, this would not be a big deal since once we build one block on the genesis block, we will always have a previous block. When testing, however, we will need to create more blocks. This will also give us more ether if our tester runs out of ether. The code to mine a block is below:

```
s.mine(n=1,coinbase=tester.a0)
```

where n refers to the number of blocks to be mined and coinbase refers to the tester address that will "do" the mining. Note that this is python code, and the s variable references the current state of the "blockchain". You can not mine from inside of a Serpent contract. This function must be used after we have create the state [?]

4.12 Gas

As we know, Ethereum smart contracts are essentially small programs. As any programmer knows, infinite loops and inefficient code can cause problems. The ethereum network is not extremely powerful, as it is only designed to execute small programs. To incentivize efficient programming, the execution of contracts requires gas. An amount of gas is "burned" for every operation that occurs in the transaction. Since a contract must be funded, this eliminates the ability for an infinite loop to occur.

When using the tester, we can simply send an arbitrary amount of gas (that is above the amount the contract needs to execute) since it is free. However, when executing contracts on an actual block chain, we need to make sure that we only spend what we need to. The best way to do this in pyethereum.tester is to use the variable $s.block.gas_used$ where s is the current state. This prints the gas used thus far in the current block. Since this is the tester, and we are the only ones putting transactions into the block, this only counts the gas used by our transactions. Let's look at an example:

```
evm_code = serpent.compile(serpent_code)
translator = abi.ContractTranslator(serpent.mk_full_signature(serpent_code))

s = tester.state()
print(s.block.gas_used)  #Call 1 = 0 gas
c = s.evm(evm_code)

print(s.block.gas_used) #Call 2 = 3650 gas

data = translator.encode('deposit', [])
o = translator.decode('deposit', s.send(tester.k0, c, 1000, data))
print(o)

print(s.block.gas_used) #Call 3 = 4641 gas
```

In the example above, we print the amount of gas used three times. At the first call, we have not added any transactions to the block chain, so we have not used any gas yet. At the second call, we have added our contract to the block chain, so we have used 3650 gas. Let's say we wanted to know how much gas the deposit command used. We can subtract the amount of gas used at call 3 from the amount of gas used at call 2 ($4641 - 3650 = 991$) to show that calling deposit with the given parameters costs 991 gas.

Now that we know the quantity of gas needed to execute the transaction, we need to figure out how much that will cost. Currently, on the public block chain, gas cost 10 szabo per unit. However, that unit will adjust when ethereum is officially released. The total price of the contract will be equal to the gas price multiplied by the gas cost of the transaction.

Note that when a transaction runs out of gas, the execution of the transaction simply rolls back - it's like it never happened. However, gas and any value sent to the contract or miner will not be refunded. [?, ?]

4.13 The Callstack

The maximum callstack in Ethereum is of size 1024. An attacker could call a contract with an already existing callstack. If a send function (or any function) is called while already at the maximum callstack size, it will create the exception, but the execution of the contract will continue. Therefore, they could cause certain portions of the contract to be skipped. To solve this, put the following code at the beginning of your functions to ensure that an attacker can not try to skip portions of the contact:

```
if self.test_callstack() != 1: return(-1)
```

Then create the function $test_callstack()$:

```
def test_callstack(): return(1)
```

This will add a function to the callstack. If an attacker tries to break the callstack by 1, it will cause the contract to not execute.

5 Simple Serpent Contract Example - Namecoin

Now that we understand the basics of Serpent's syntax, lets do a couple of examples to show how all of these pieces work together. First, we will make a contract that is normally called "namecoin". Essentially, it allow for us to create a basic key-value store. A key value store is a data storage structure that allows for us to associate a key with a value, and look up values based on their keys. This contract will have two different functions to call. The first is the key-value registration function and the second is a function that retrieves a value associated with a provided key.

The first function we will look at is $register(key, value)$, which takes a key and value and associates them with each other:

```
def register(key, value):
        if not self.storage[key]:
                self.storage[key] = value
                return(1)
        else:
                return(-1)
```

Lets break this down. This contract essentially consists of an if-else statement. First, we check to see if the key-value is already in storage. We can use the not statement to check if nothing is stored. So if nothing is stored, we will store the value in the persistant key-value store $self.storage[]$. However, what if the key is already taken? We can't just overwrite someone else's key! So, we just return -1.

Now that we know how to store values, we need to be able to retrieve them. For that we use the $get(key)$ function:

```
def get(key):
        if not self.storage[key]:
                return(-1)
        else:
                return(self.storage[key])
```

This function will simply return the value associated with the key. This function is very similar to our storage function. However, this time we don't store anything. If there is nothing associated with the key, we return -1. Otherwise, we return the value that is associated with the key.

The complete code for namecoin is below:

```
def register(key, value):
        if not self.storage[key]:
                self.storage[key] = value
                return(1)
        else:
                return(-1)
```

```
def get(key):
        if not self.storage[key]:
                return(-1)
        else:
                return(self.storage[key])
```

6 Basic Serpent Contract Example - Easy Bank

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 contractual currency and not Ether (which we will discuss 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():
        #Initializaze the contract creator with 10000 fake dollars
        self.storage[msg.sender] = 10000

def send_currency_to(value, destination):
        #If the sender has enough money to fund the transaction, complete it
        if self.storage[msg.sender] >= value:
                self.storage[msg.sender] = self.storage[msg.sender]  - value
                self.storage[destination] = self.storage[destination] + value
                return(1)
        return(-1)

def balance_check(addr):
        #Balance Check
        return(self.storage[addr])
```

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():
        #Initialiaze the contract creator with 10000 fake dollars
        self.storage[msg.sender] = 10000
```

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, initializes the contract creator's account with a balance of 10,000 dollars. 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 10,000 at key msg.sender (we'll identify what this is in a moment).

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, destination):
        #If the sender has enough money to fund the transaction, complete it
        if self.storage[msg.sender] >= value:
                self.storage[msg.sender] = self.storage[msg.sender]  - value
                self.storage[destination] = self.storage[destination] + value
                return(1)
        return(-1)
```

The *send_currency_to* function takes in two parameters. The first is the value in dollars that we are sending. The second is the public key of the address we are sending it to.

First, we check that the person trying to transfer money has enough in their account to successfully complete the transfer. If they do, we complete the transaction by removing the value from the sender's account and adding to the destination's account, and we return 1. If they do not have enough money, we simply return -1, denoting that the transaction failed.

The *balance_check* function simply returns the value currently stored in the provided public key's account.

Great! We have officially worked our way through a very basic contract example! 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?

[?]

7 Moderate Serpent Contract Example - Bank

Let's take a quick look at a smart contract that impelements a bank. 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 bank further. Our bank will allow users to store Ether in units of Wei. It must be capable of the following actions allowing users to:

1. Deposit money into their account.
2. Transfer money from their account to another account.
3. Withdraw their money.
4. Check their balance.

```
def deposit():
        if not self.storage[msg.sender]:
                self.storage[msg.sender] = 0
        self.storage[msg.sender] += msg.value
        return(1)
def withdraw(amount):
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                self.storage[msg.sender] -= amount
                send(0, msg.sender, amount)
                return(1)
def transfer(amount, destination):
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                if not self.storage[destination]:
                        self.storage[destination] = 0
                self.storage[msg.sender] -= amount
                self.storage[destination] += amount
                return(1)
def balance():
        if not self.storage[msg.sender]:
                return(-1)
        else:
                return(self.storage[msg.sender])
```

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

```
def deposit():
        if not self.storage[msg.sender]:
                self.storage[msg.sender] = 0
        self.storage[msg.sender] += msg.value
        return(1)
```

This method is a relatively simple method. It allows for a user to deposit funds into their account. Similar to our Namecoin example, we are using $self.storage[]$ so we can associate the address of the person who owns the account with the value of the ether they are storing in their account. We do this on the third and fourth lines, where we use *msg.sender* as the key. *msg.sender* stores the address of whomever sent the command. The other built-in variable reference we use is *msg.value*. This stores the amount of ether (measured in wei) that is sent with the transaction. When ether is sent with a command to a contract, it is stored by the contract. Therefore, we just need to account for how much each person owes. This is stored as the value we are associating with the key in $self.storage[]$.

This method first checks if the person already has an account. If they don't, we make one for them and give it a zero balance. Next, it adds the value sent with the deposit to the person's account (stored in $self.storage[]$). Then, it returns 1. Since something will always be deposited, there isn't really an error condition that can occur (where we may return something else).

```
def withdraw(amount):
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                self.storage[msg.sender] -= amount
                send(0, msg.sender, amount)
                return(1)
```

This method here is doing essentially the opposite of the of the deposit method. Here we are taking *amount* ether out of our account and sending it to ourself. First, we check to make sure they have enough to withdraw. If we don't, we return -1. We could technically return anything, but in this guide, we use negative numbers to symbolize that there is an error. If they do have enough wei in there account, we simply subtract that from that from their account (still using *msg.sender* as a key). However, how do we send that wei back to the account owner? Simple! We simply use the send function. The send function takes three parameters. First, it takes the amount of gas we are sending with the contract. Since we are going to assume that this is being refunded to a user and not another contract, we don't need to send any gas with it. The next parameter is the address that we are sending this money to. Since *msg.sender* own the account, we are going to send this ether back to msg.sender. Next, since we have shown that there is at least the requested amount in the account, we will send that amount to them. Finally, we will return 1 to show that the operation completed successfully.

Finally, lets look at our transfer method:

```
def transfer(amount, destination):
        if self.storage[msg.sender] < amount:
                return(-1)
        else:
                if not self.storage[destination]:
                        self.storage[destination] = 0
                self.storage[msg.sender] -= amount
                self.storage[destination] += amount
                return(1)
```

This method allows for someone to move ether from one account to another. It works very similarly to the deposit and withdraw methods we have already looked at. The only difference with this one is that one of the parameters is called "destination". This parameter takes in the public address of the person's account we are sending the money to. Remember that we use the public address as the key in our *self.storage*[] key-value store.

In this method, we first check to make sure there is enough money in the account. If there is, we check if the destination account exists. If it doesn't, we make one. Finally, we transfer the funds between the accounts.

I will leave it as an exercise to you to see how the balance function works.

8 Student Exercise - Mutual Credit System

Now that you have looked at a few examples of ethereum contracts, it's time for you to try it for yourself. We are going to continue with the idea of a banking contract, but we are going to change it up. We want you set up what we are a calling a "Mutual Credit System". In this system, everyone will start off with a balance of zero. When you make a transaction, you pay using debt, so your balance becomes negative. The person you pay gains credit, so his balance becomes positive. After all of the transactions, people will have varying amounts of money, some positive, some negative. We are limiting the amount of debt one is allowed to spend to 1000 credits. Note that we will be using our own currency, not ether.

To complete this task you will need to use *self.storage*[] for persistent storage. You will need to create two methods. The first "transfer" which accepts a public key and a value. This will transfer the credits from the *msg.sender*'s account to the public key's account (return 0). If the account that is sending the credits will exceed 1000 credits of debt, the transaction should be declined (return -1).

You will also need to implement a balance method that takes in the public key the sender wants the balance of, and returns the balance of that public key.

For more information on Mutual Credit Systems, visit `http://p2pfoundation.net/Mutual_Credit`.

9 Resource Overview

This guide is provided as a ”one stop shop” for a quick way to learn how to program smart contracts with ethereum. However, the platform is always changing and it would be impossible for this guide to cover everything. We have provided some links below that provide some additional insight into programming ethereum contracts. All of these sources were actually used in creating this guide.

- Ethereum Wiki - `https://github.com/ethereum/wiki/wiki` - This source has some fantastic tutorials and reference documentation about the underlying systems that power Ethereum. This should be your first stop when you have problems with Ethereum.
- Serpent Tutorial - `https://github.com/ethereum/wiki/wiki/Serpent` - This is the official serpent tutorial that is on the Ethereum Wiki. It gives a good, brief overview of many of the most used components of serpent and goes over basic testing.
- KenK's Tutorials - Most of these tutorials use old versions of Serpent, but should be updated soon. These give a great overview of some of Ethereum's more advanced features. Note that these tutorials use cpp-ethereum and not pyethereum.
  - Part 1: `http://forum.ethereum.org/discussion/1634/tutorial-1-your-first-contract`
  - Part 2: `http://forum.ethereum.org/discussion/1635/tutorial-2-rainbow-coin`
  - Part 3: `http://forum.ethereum.org/discussion/1636/tutorial-3-introduction-to-the-javascript-api`