Lab: Step by Step towards Programming a Safe Smart Contract

Kevin Delmolino
del@terpmail.umd.edu

Mitchell Arnett
marnett@umd.edu

Ahmed Kosba
ahmed.essamk@gmail.com

Andrew Miller
amiller@cs.umd.edu

Elaine Shi
elaine@cs.umd.edu

Contents

1 Introduction

Intended audience. This tutorial is intended for instructors who wish to conduct a smart contract programming lab, or students/developers who want to learn about smart contract programming.

Applicability of this tutorial. While we use Ethereum's Serpent language as a specific example, this document is intended to be broadly applicable to smart contract programming on top of a decentralized cryptocurrency.

Loosely speaking, we shall make some simplifying assumptions about the security model. We assume that the decentralized cryptocurrency under consideration has a secure and incentive compatible consensus protocol, such that when rational miners comprise the majority of compute power (or other forms of resources), in a Nash equilibrium, it is in the best interest of rational miners to honestly execute a contract's program logic. In reality, existing decentralized cryptocurrency achieves only heuristic security. But we will make this assumption nevertheless. How to design a provably secure decentralized consensus protocol under rationality assumptions is a topic of future research.

Conceptual model for programming smart contracts. Based on the above simplifying assumptions, we can simply assume that a smart contract is a special party in the system who is trusted for correct operation, but not for privacy. Specifically, messages and money sent to and from the contract, as well as states stored by the contract are *publicly* visible. If you are a cryptographer reading this, it is meaningful to compare a contract with a cryptographic ideal functionality – the latter is trusted both for correctness and privacy.

Accompanying materials. All example smart contracts in this document are written for a specific snapshot of Ethereum. For the users' convenience, we offer a VM image with appropriate versions of the software pre-installed [1]. We also provide detailed Ethereum reference manuals geared towards this specific snapshot of Serpent [2]. Finally, we also recommend the reader to a more concise, Powerpoint presentation of this tutorial by Elaine Shi and Andrew Miller [3].

For best learning outcomes, we strongly recommend that the reader take a hands-on approach towards reading this lab tutorial, by running our examples in `py-ethereum`, and deploying fixes to the pitfalls that we point out in this document.

2 A Rock-Paper-Scissors Example

We use a very simple Rock-Paper-Scissors contract to illustrate several pitfalls of smart contract programming. Mistakes resembling these were actually observed in our Smart Contract Programming Lab in "CMSC 414 - Undergraduate Security".

2.1 Corner Cases in Coding State Machines

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 in the *add_player* contract below:

```
def add_player():
        if not self.storage["player1"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] =
                                self.storage["WINNINGS"] + msg.value
                        self.storage["player1"] = msg.sender
                        return(1)
                return (0)
        elif not self.storage["player2"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] =
                                self.storage["WINNINGS"] + msg.value
                        self.storage["player2"] = msg.sender
                        return(2)
                return (0)
        else:
                return(0)
```

In this section a user adds themselves to the game by sending a small amount of Ether with their 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 scenarios our contract currently allows 1) a potential entrant sends too much or too little Ether, 2) there are already two participants, so additional players send transactions to join, but are not allowed. In both of the following scenarios the contract will keep their money. If someone sent too much or too little to enter they will not be added as a player, but their funds will be kept. Even worse, if the match is full any person who tries to join (they have no way of knowing it is full) will pay to play but never be added to a game! Both of these errors will cause distrust in our contract, eventually resulting in the community not trusting this particular contract and, more importantly, this contract's author - you.

So how do we fix these issues? It seems like our contract needs the ability to refund - 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? Look at the previous section ”Sending Wei” for more information.

2.2 Implementing Cryptography

It goes without saying that as a student in a computer security course you would implement cryptographic practices wherever you can. Thus given a contract that requires impactful user inputs (ones that affect the outcome of said contract) cryptography 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 and think about possible vulnerabilities:

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

We can see that our *input*() function identifies the sender with *msg.sender* and then stores their input *choice* in plaintext (where *choice* = 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 the winning choice to ensure they always win the prize pool. This can be fixed by using a commitment scheme. We will alter *input*() to accept a hash of [sender, choice, and a nonce]. After both players have committed their inputs they will send their *choice* and *nonce* (as plaintext) to an *open*() function. *open*() will verify what they sent to *input*(). What they send to *open*() will be hashed, and that hash will be checked against the hash the user committed through *input*(). If the two hashes don't match then the player will automatically lose based on the assumption they were being dishonest. Understanding where crypto elements should be used is crucial to justifying why others should use your contract.

In order to enhance the security and fairness of our contract we will implement a commitment scheme using the hashing functions discussed earlier in this guide. The first change that is necessary in our contract is to have the *input*() 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 *choice* = SHA3(msg.sender's public address, 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 based on their committed choice alone.

Now that we have the hash stored in the contract we need to implement an *open*() function that we discussed earlier. Our *open*() function will take the plaintext inputs and nonces from the players as parameters. We will hash these together with the unique sender ID and compare to the stored hash to verify that they claim to have committed as their input is true. 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 *choice* sent to *input*() was. Below is the updated, cleaned up contract (version2.py) implementing an *open*() and modifying *check*() to work with our new scheme. Notice we have added a method *open*() and reorganized our *check*():

```
def input(player_commitment):
        if self.storage["player1"] == msg.sender:
                self.storage["p1commit"] = player_commitment
                return (1)
        elif self.storage["player2"] ==  msg.sender:
                self.storage["p2commit"] = player_commitment
                return(2)
        else:
                return(0)

def open(choice, nonce):
        if self.storage["player1"] == msg.sender:
                if sha3([msg.sender, choice, nonce], items=3) == self.storage["p1commit"]:
                        self.storage["p1value"] = choice
                        self.storage["p1reveal"] = 1
                        return(1)
                else:
                        return(0)
        elif self.storage["player2"] == msg.sender:
                if sha3([msg.sender, choice, nonce], items=3) == self.storage["p2commit"]:
                        self.storage["p2value"] = choice
                        self.storage["p2reveal"] = 1
                        return(2)
                else:
                        return(0)
        else:
                return(-1)

def check():
        #check to see if both players have revealed answer
        if self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                #If player 1 wins
                if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
                        send(100,self.storage["player1"], self.storage["WINNINGS"])
                        return(1)
                #If player 2 wins
                elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
                        send(100,self.storage["player2"], self.storage["WINNINGS"])
                        return(2)
                #If no one wins
                else:
                        send(100,self.storage["player1"], 1000)
                        send(100,self.storage["player2"], 1000)
                        return(0)
        #if p1 revealed but p2 did not, send money to p1
        elif self.storage["p1reveal"] == 1 and not self.storage["p2reveal"] == 1:
                send(100,self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        #if p2 revealed but p1 did not, send money to p2
        elif not self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                send(100,self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        #if neither p1 nor p2 revealed, keep both of their bets
        else:
                return(-1)
```

2.3 Incentive Compatability

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 within a time frame to avoid cheating. Let's look and see how our RPS contract holds up with regard to incentives:

```
def check():
        #check to see if both players have revealed answer
        if self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
                #If player 1 wins
                if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
                        send(100,self.storage["player1"], self.storage["WINNINGS"])
```

```
                    return(1)
                #If player 2 wins
                elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
                    send(100,self.storage["player2"], self.storage["WINNINGS"])
                    return(2)
                #If no one wins
                else:
                    send(100,self.storage["player1"], 1000)
                    send(100,self.storage["player2"], 1000)
                    return(0)
        #if p1 revealed but p2 did not, send money to p1
        elif self.storage["p1reveal"] == 1 and not self.storage["p2reveal"] == 1:
            send(100,self.storage["player1"], self.storage["WINNINGS"])
            return(1)
        #if p2 revealed but p1 did not, send money to p2
        elif not self.storage["p1reveal"] == 1 and self.storage["p2reveal"] == 1:
            send(100,self.storage["player2"], self.storage["WINNINGS"])
            return(2)
        #if neither p1 nor p2 revealed, keep both of their bets
        else:
            return(-1)
```

Given the version at the end of this section our contract is *almost* 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. This requires one player to spend gas to check to see who won, while the other player doesn't need to spend the same amount. There is currently no way to require two people to spend equal amount of gas to call one function. How could this affect the incentives of the contract?

In the next section we will look at how the current block number and the amount of blocks that have passed affect the security of a contract. We will look to alter our contract further so if someone doesn't open (verify) their rock/paper/scissors within a given timeframe (i.e. 5 blocks after they are added to the contract), the contract would, by default, send the money to the person who *did* verify their input by the deadline. This incentivizes both users to verify their inputs before the *check*() function is called after a random amount of blocks have been published; if you don't verify you are *guaranteed* to lose.

2.4 Original Buggy Rock, Paper, Scissor Contract

```
data winnings_table[3][3]

def init():
        #If 0, tie
        #If 1, player 1 wins
        #If 2, player 2 wins

        #0 = rock
        #1 = paper
        #2 = scissors

        self.winnings_table[0][0] = 0
        self.winnings_table[1][1] = 0
        self.winnings_table[2][2] = 0

        #Rock beats scissors
        self.winnings_table[0][2] = 1
        self.winnings_table[2][0] = 2

        #Scissors beats paper
        self.winnings_table[2][1] = 1
        self.winnings_table[1][2] = 2

        #Paper beats rock
        self.winnings_table[1][0] = 1
        self.winnings_table[0][1] = 2

        self.storage["MAX_PLAYERS"] = 2
        self.storage["WINNINGS"] = 0

def add_player():
        if not self.storage["player1"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                        self.storage["player1"] = msg.sender
                        return(1)
                return (0)
        elif not self.storage["player2"]:
                if msg.value == 1000:
                        self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                        self.storage["player2"] = msg.sender
                        return(2)
                return (0)
        else:
                return(0)
```

```

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

def check():
        #If player 1 wins
        if self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 1:
                send(100,self.storage["player1"], self.storage["WINNINGS"])
                return(1)
        #If player 2 wins
        elif self.winnings_table[self.storage["p1value"]][self.storage["p2value"]] == 2:
                send(100,self.storage["player2"], self.storage["WINNINGS"])
                return(2)
        #If no one wins
        else:
                send(100,self.storage["player1"], self.storage["WINNINGS"]/2)
                send(100,self.storage["player2"], self.storage["WINNINGS"]/2)
                return(0)

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

Implement the changes from each of the aboved sections to have a much stronger contract!

References

[1] Lab virtual machine image. `http://www.cs.umd.edu/~elaine/smartcontract/coursematerial.html`.

[2] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. A programmer's guide to ethereum and serpent. `http://www.cs.umd.edu/~elaine/smartcontract/coursematerial.html`.

[3] Elaine Shi and Andrew Miller. Undergraduate ethereum lab at maryland and insights gained. `https://docs.google.com/presentation/d/1esw_lizWGO6zrWaOQKcbwrySM4K9KzmRD3rtBUxOzEw/edit?usp=sharing`", 2015.