Kevin Zhao
Sep 5th

Problem 1. Answer the following questions about the transaction where you received the bitcoin. If you received more than one transfer, include all the transaction IDs).

a. What is the transaction ID?

4d894aa7bc7e8c012ab00348adb356a573ca8c20657156b8755d691c5475b6e7

b. What was the transaction fee for the transaction? (Give your answer in BTC, as well as current approximate US dollar value.)

0.0001BTC and 0.02 dollars.

c. What was the total value of all the transactions in the block containing your transfer? (Note: https://blockchain.info provides this info conveniently, although you could compute it yourself)

1,755.15212027 BTC

d. How long did it take from when the transaction was received until it had 3 confirmations? (Include an explanation of how you estimated this in your answer.)

It's been approximately 8 days 19 hours since I received my coin, and there are 1385 confirmations. I used the answer to (total hour) / (total confirmations) to get the answer of 9.14 minutes per confirmation. That' means 3 confirmation take 27.4 minutes.

Problem 2. See how much can you figure out about the way bitcoin was transferred to students in the class, starting from your transactions.

a. Identify the bitcoin addresses of what are likely to be other students in the class (you could potentially find all of them, but it is enough to find 3).

Student 1: 1DkHaFKyrXhE28KsybBntCxoYeSoBumPfU
Student 2: 1GMrGqvF8FwbgCSShCdzijcSUHzmNR1VCz
Student 3: 1kKvKdXcLvyHJTFuNxkdxm2tEnkG8hB1S

b. Trace back the source of the bitcoin as far as you can. Bonus points if you can figure out from which exchange the bitcoin was purchased and when.

It seems that the account that gave out small amount of bitcoins to students received 0.2 BTC from 14J6ep326owXDpc1waViGJNB1onSFy9eou address. Which I believe is the source of most of the bitcoin that was distributed in class. By digging deeper into this specific transaction(66809ee9b41310b53e3fc94cbff8fe7c870d6df4ca7561614137311f651ed9a8), it seems that it was relayed by IP address in NYC. I would take a guess and say that it's bought on ItBit because ItBit is based in NYC.

c. (Bonus) Can you learn anything about where the send of the bitcoin is located geographically? (In this case, you have external information to know I'm in Charlottesville, but what could you learn about the sender's probable location just from the information in the blockchain?)

By looking at the time of the transaction, I can see that the sender is probably lives on the East coast or maybe Central United States, since some transaction took place around 8pm and 9pm EST and 3pm EST. These times will be around 11pm/12am for PST, and 6pm PST, these times are not likely time that people would want to be doing Bitcoin transaction. People are eating dinner at 6pm and might be getting to bed around 11pm/12am.

Problem 3. Suppose a malicious developer wanted to distribute a bitcoin wallet implementation that would steal as much bitcoin as possible from its users with a little chance as possible of getting caught.

(a) Explain things a malicious developed might do to create an evil wallet.

The developer could upload the users' private keys to a remote server without telling users. Then he could write a script to run on the server and slowly take bitcoins away from the users' wallets. If he/she steals too much at once, it will be really easy to be detected, but by being patient, he/she can slowly steal a lot of money from users in small chunks.

(b) How confident are you your money is safe in the wallet you are using, and what would you do to increase your confidence if you were going to store all of your income in it?

Not really confident since I don't know the exact architecture of my wallet app, can't be certain that there isn't a backdoor that uploads my private key to a remote server. I would monitor my Internet traffic when I am using the application to see if there is anything that's sending others my private key. I will also need a third party to verify that this implementation is secure, also someone to constantly protect my laptop to prevent others to hack my computer and steal the private key.

Problem 4. Verify that the modulus used as secp256k1.P inbtcec.go is correct. You can do this either using math/big, Go's bit integer library to do computations on such large numbers, or by computing it by hand. (For your answer, just show how you verified the modulus. Including a snippet of code is fine.)

```
package main

import (
  "fmt"
  "math/big"
  )

func main() {
  a := big.NewInt(2)
  b := big.NewInt(256)
  c := big.NewInt(32)
  d := big.NewInt(9)
  e := big.NewInt(8)
  f := big.NewInt(7)
  g := big.NewInt(6)
  h := big.NewInt(4)
  i := big.NewInt(0)
  p := big.NewInt(0)
  j :=
fromHex("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC
2F")
  p.Sub(powerFunc(a, b), powerFunc(a, c))
  p.Sub(p, powerFunc(a, d))
  p.Sub(p, powerFunc(a, e))
  p.Sub(p, powerFunc(a, f))
```

```
  p.Sub(p, powerFunc(a, g))
  p.Sub(p, powerFunc(a, h))
  p.Sub(p, powerFunc(a, i))
  fmt.Println(p)
  fmt.Println(j)
  fmt.Println(p.Cmp(j) == 0)
}

func fromHex(s string) *big.Int {
  r, ok := new(big.Int).SetString(s, 16)
  if !ok {
    panic("invalid hex in source file: " + s)
  }
  return r
}

func powerFunc(i *big.Int, p *big.Int) *big.Int {
  x := big.NewInt(0)
  x.Exp(i,p,nil)
  return x
}
```

Problem 5:
I need to know that when the code is run, the private key is not sent to anyone, or saved to any part of my computer. I also need to confirm that the implementation of the generation of the keys is exactly as specified by the Bitcoin standard.

Problem 6:

```
package main

import (
  "fmt"
  "encoding/hex"
  "github.com/btcsuite/btcd/btcec"
  "regexp"
  "log"
  "github.com/btcsuite/btcd/chaincfg"
  "github.com/btcsuite/btcutil"
  )

func generateKeyPair() (*btcec.PublicKey, *btcec.PrivateKey) {

  priv, err := btcec.NewPrivateKey(btcec.S256())
  if err != nil {
    log.Fatal(err)
  }
```

```
  return priv.PubKey(), priv
}

func generateAddr(pub *btcec.PublicKey) *btcutil.AddressPubKeyHash {

  net := &chaincfg.MainNetParams
  b := btcutil.Hash160(pub.SerializeCompressed())
  addr, err := btcutil.NewAddressPubKeyHash(b, net)
  if err != nil {
    log.Fatal(err)
  }

  return addr
}

func generateVanityAddress(pattern string) (*btcec.PublicKey, *btcec.PrivateKey){
  flag := false
  pub , private := generateKeyPair()
  for flag == false {
    pub_key , private_key := generateKeyPair()
    addr := generateAddr(pub_key)
    match, error := regexp.MatchString(pattern, addr.String())
    if error != nil{
      log.Fatal(error)
    }
    if match {
      fmt.Printf(addr.String() + "\n")
      return pub_key, private_key
    }
  }
  return pub, private
}

func main() {
  fmt.Print("Enter text: ")
  var input string
  fmt.Scanln(&input)
  // pub, priv := generateVanityAddress("3.*1.*4.*1.*5.*9.*")
  pub, priv := generateVanityAddress(input)
  fmt.Printf("This is a private key in hex:\t[%s]\n",
    hex.EncodeToString(priv.Serialize()))

  fmt.Printf("This is a public key in hex:\t[%s]\n",
    hex.EncodeToString(pub.SerializeCompressed()))
}
```

Problem 7:
My vanity address has “kevz” in it, which is a version of my name:
1BDfjj27P8tarosnbU8UF85kevz2TvudeY

Problem 8:
No, vanity addresses are not more secure than another vanity address as it takes the same amount of time and effort to decrypt.

Problem 9:
Transaction ID: 7424a92a1c49887137bfe16c120f0e2989b9612bf9daf41034f737c9f71535d3

Problem 10:
Done.

Problem 11:

```
/*
** spend.go
**
** cs4501 Fall 2015
** Problem Set 1
*/

/*
For this program to execute correctly the following needs to be provided:

- An internet connection
- A private key
- A receiving address
- The raw json of the funding transaction
- The index into that transaction that funds your private key.

The program will formulate a transaction from these provided parameters
and then it will dump the newly formed tx to the command line as well as
try to broadcast the transaction into the bitcoin network. The raw hex dumped
by the program can parsed into a 'semi' human readable format using services
like: https://blockchain.info/decode-tx

*/

package main

import (
        "bytes"
```

```
        "encoding/hex"
        "encoding/json"
        "flag"
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "os"

        "github.com/btcsuite/btcd/btcec"
        "github.com/btcsuite/btcd/chaincfg"
        "github.com/btcsuite/btcd/txscript"
        "github.com/btcsuite/btcutil"
        "github.com/btcsuite/btcd/wire"
)

var toaddress = flag.String("toaddress", "", "The address to send Bitcoin to.")
var privkey = flag.String("privkey", "", "The private key of the input tx.")
var txid = flag.String("txid", "", "The transaction id corresponding to the funding Bitcoin
transaction.")
var vout = flag.Int("vout", -1, "The index into the funding transaction.")
var amount = flag.Int("amount", 0, "The amount of Bitcoin you want to transfer.")

// Use 10,000 satoshi as the standard transaction fee.
const TX_FEE = 10000

type requiredArgs struct {
        txid      *wire.ShaHash
        vout      uint32
        toAddress btcutil.Address
        privKey   *btcec.PrivateKey
        amount    uint32
}

// getArgs parses command line args and asserts that a private key and an
// address are present and correctly formatted.
func getArgs() requiredArgs {
        flag.Parse()
        if *toaddress == "" || *privkey == "" || *txid == "" || *vout == -1 || *amount == 0 {
                fmt.Println("\nThis program generates a bitcoin trans action that moves coins
from an input to an output.\n" +
                        "You must provide a key, a receiving address, a transaction id (the hash\n"
+
                        "of a tx) and the index into the outputs of that tx that fund your\n" +
                        "address. Use http://blockchain.info/pushtx to send the raw
transaction.\n")
```

```
                flag.PrintDefaults()
                fmt.Println("")
                os.Exit(0)
        }

        pkBytes, err := hex.DecodeString(*privkey)
        if err != nil {
                log.Fatal("problem with private key")
                log.Fatal(err)
        }

        // PrivKeyFromBytes returns public key as a separate result, but can ignore it here.
        key, _ := btcec.PrivKeyFromBytes(btcec.S256(), pkBytes)

        addr, err := btcutil.DecodeAddress(*toaddress, &chaincfg.MainNetParams)
        if err != nil {
                log.Fatal("problem with address")
                log.Fatal(err)
        }

        txid, err := wire.NewShaHashFromStr(*txid)
        if err != nil {
                log.Fatal("problem with txid")
                log.Fatal(err)
        }

        args := requiredArgs{
                txid:      txid,
                vout:      uint32(*vout),
                toAddress: addr,
                privKey:   key,
                amount:                  uint32(*amount),
        }

        return args
}

type BlockChainInfoTxOut struct {
        Value     int    `json:"value"`
        ScriptHex string `json:"script"`
}

type blockChainInfoTx struct {
        Ver     int                 `json:"ver"`
        Hash    string              `json:"hash"`
        Outputs []BlockChainInfoTxOut `json:"out"`
```

```
}

// Uses the txid of the target funding transaction and asks blockchain.info's
// api for information (in json) relaated to that transaction.
func lookupTxid(hash *wire.ShaHash) *blockChainInfoTx {

	url := "https://blockchain.info/rawtx/" + hash.String()
	resp, err := http.Get(url)
	if err != nil {
		log.Fatal(fmt.Errorf("Tx Lookup failed: %v", err))
	}

	b, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		log.Fatal(fmt.Errorf("TxInfo read failed: %s", err))
	}

	//fmt.Printf("%s\n", b)
	txinfo := &blockChainInfoTx{}
	err = json.Unmarshal(b, txinfo)
	if err != nil {
		log.Fatal(err)
	}

	if txinfo.Ver != 1 {
		log.Fatal(fmt.Errorf("Blockchain.info's response seems bad: %v", txinfo))
	}

	return txinfo
}

// getFundingParams pulls the relevant transaction information from the json returned by
blockchain.info
// To generate a new valid transaction all of the parameters of the TxOut we are
// spending from must be used.
func getFundingParams(rawtx *blockChainInfoTx, vout uint32, amount uint32) (*wire.TxOut,
*wire.OutPoint) {
	blkChnTxOut := rawtx.Outputs[vout]

	hash, err := wire.NewShaHashFromStr(rawtx.Hash)
	if err != nil {
		log.Fatal(err)
	}

	amnt := (btcutil.Amount(int64(0)))
	// Then convert it to a btcutil amount
```

```
	if ((btcutil.Amount(int64(amount))) > (btcutil.Amount(int64(blkChnTxOut.Value)) -
10000)) {
		log.Fatal("The amount is higher than what the person owns.")
	} else {
		amnt = (btcutil.Amount(int64(amount)))
	}

	if err != nil {
		log.Fatal(err)
	}

	outpoint := wire.NewOutPoint(hash, vout)

	subscript, err := hex.DecodeString(blkChnTxOut.ScriptHex)
	if err != nil {
		log.Fatal(err)
	}

	oldTxOut := wire.NewTxOut(int64(amnt), subscript)

	return oldTxOut, outpoint
}

func main() {
	// Pull the required arguments off of the command line.
	reqArgs := getArgs()

	// Get the bitcoin tx from blockchain.info's api
	rawFundingTx := lookupTxid(reqArgs.txid)

	// Get the parameters we need from the funding transaction
	oldTxOut, outpoint := getFundingParams(rawFundingTx, reqArgs.vout, reqArgs.amount)

	// Formulate a new transaction from the provided parameters
	tx := wire.NewMsgTx()

	// Create the TxIn
	txin := createTxIn(outpoint)
	tx.AddTxIn(txin)

	// Create the TxOut
	txout := createTxOut(oldTxOut.Value, reqArgs.toAddress)
	tx.AddTxOut(txout)

	// Generate a signature over the whole tx.
	sig := generateSig(tx, reqArgs.privKey, oldTxOut.PkScript)
```

```
	tx.TxIn[0].SignatureScript = sig

	// Dump the bytes to stdout
	dumpHex(tx)

	// Send the transaction to the network
	broadcastTx(tx)
}

// createTxIn pulls the outpoint out of the funding TxOut and uses it as a reference
// for the txin that will be placed in a new transaction.
func createTxIn(outpoint *wire.OutPoint) *wire.TxIn {
	// The second arg is the txin's signature script, which we are leaving empty
	// until the entire transaction is ready.
	txin := wire.NewTxIn(outpoint, []byte{})
	return txin
}

// createTxOut generates a TxOut that can be added to a transaction.
func createTxOut(inCoin int64, addr btcutil.Address) *wire.TxOut {
	// Pay the minimum network fee so that nodes will broadcast the tx.
	outCoin := inCoin - TX_FEE
	// Take the address and generate a PubKeyScript out of it
	script, err := txscript.PayToAddrScript(addr)
	if err != nil {
		log.Fatal(err)
	}
	txout := wire.NewTxOut(outCoin, script)
	return txout
}

// generateSig requires a transaction, a private key, and the bytes of the raw
// scriptPubKey. It will then generate a signature over all of the outputs of
// the provided tx. This is the last step of creating a valid transaction.
func generateSig(tx *wire.MsgTx, privkey *btcec.PrivateKey, scriptPubKey []byte) []byte {

	// The all important signature. Each input is documented below.
	scriptSig, err := txscript.SignatureScript(
		tx,                 // The tx to be signed.
		0,                  // The index of the txin the signature is for.
		scriptPubKey,       // The other half of the script from the PubKeyHash.
		txscript.SigHashAll,  // The signature flags that indicate what the sig covers.
		privkey,            // The key to generate the signature with.
		true,               // The compress sig flag. This saves space on the blockchain.
	)
	if err != nil {
```

```
                log.Fatal(err)
        }

        return scriptSig
}

// dumpHex dumps the raw bytes of a Bitcoin transaction to stdout. This is the
// format that Bitcoin wire's protocol accepts, so you could connect to a node,
// send them these bytes, and if the tx was valid, the node would forward the
// tx through the network.
func dumpHex(tx *wire.MsgTx) {
        buf := bytes.NewBuffer(make([]byte, 0, tx.SerializeSize()))
        tx.Serialize(buf)
        hexstr := hex.EncodeToString(buf.Bytes())
        fmt.Println("Here is your raw bitcoin transaction:")
        fmt.Println(hexstr)
}

type sendTxJson struct {
        RawTx string `json:"rawtx"`
}

// broadcastTx tries to send the transaction using an api that will broadcast
// a submitted transaction on behalf of the user.
//
// The transaction is broadcast to the bitcoin network using this API:
//    https://github.com/bitpay/insight-api
//
func broadcastTx(tx *wire.MsgTx) {
        buf := bytes.NewBuffer(make([]byte, 0, tx.SerializeSize()))
        tx.Serialize(buf)
        hexstr := hex.EncodeToString(buf.Bytes())

        url := "https://insight.bitpay.com/api/tx/send"
        contentType := "application/json"

        fmt.Printf("Sending transaction to: %s\n", url)
        sendTxJson := &sendTxJson{RawTx: hexstr}
        j, err := json.Marshal(sendTxJson)
        if err != nil {
                log.Fatal(fmt.Errorf("Broadcasting the tx failed: %v", err))
        }
        buf = bytes.NewBuffer(j)
        resp, err := http.Post(url, contentType, buf)
        if err != nil {
                log.Fatal(fmt.Errorf("Broadcasting the tx failed: %v", err))
```

```
	}

	b, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		log.Fatal(err)
	}

	fmt.Printf("The sending api responded with:\n%s\n", b)
}
```

Problem 12:

I tried to double spend by sending two requests one after another within 10 seconds, and that didn't work.

After doing some research, it seems that the transactions are broadcasted to the miners, so if I broadcast from the same place, it's likely the same miner will get it, then the double spend will be rejected immediately. On the other hand, if I broadcast the second double spend transaction somewhere far away, which means another miner will pick that up, and it might not be rejected immediately.

I have a EC2 instance setup, but I don't have enough bitcoin on my account since a lot of it was taken away by transaction fee because I tried to transfer too little an mount to my vanity address and trying to move only a little money back to my MultiBit wallet. I will try this when I get more Bitcoin in the future.