Reid Bixler
rmb3yz
CS 4501
9/15/15

Problem 1

a. What is the transaction ID?

7e0ea9d41b45c5264c62b0863f5394c3ea918eae4fa50b5267e512e0e9cf452a

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

0.0001 BTC ~ 0.02 USD

c. What was the total value of all the transactions in the block containing your transfer?

7,531.39109632 BTC ~ 1,735,157.19 USD

d. How long did it take from when the transaction was received until it had 3 confirmations?

The time at which it was received was 2015-08-28 15:10:41
The time at which it was included in the block was 2015-08-28 15:17:16
There were a total number of 2688 Confirmations for the transaction
Assuming a constant rate of confirmations in the 395 seconds between reception and inclusion, confirmations were coming at a rate of 6.8 per second. This means that it took about 0.44 seconds to get 3 confirmations.

Problem 2

a. Identify the bitcoin addresses of what are likely to be other students in the class.

12RZPErCQfacy2Rs8VTpBZkRuSKEH5HCTG
14yyZsLZZn7qkvMfvLBD7cJvuYXytn2bdQ
112Kh8RHajxsj2yqdvrdcq5L4bLm8xymY (Soon to be sent to student)
1FM45Varjz955Sh9SrEp41XH3gRTcFM4i6
1G8Dbnesf35V7gKRuc3Cv4EeGCwWqMdXkE (Mine!)
1KEz1rFhEoy2vQV4zQcUCFWPYy7EkJyRcM
15X2qKbsXxPHbgfFTc6UgXmETVe7fpBUzX
1DkHaFKyrXhE28KsybBntCxoYeSoBumPfU
1GMrGqvF8FwbgCSShCdzijcSUHzmNR1VCz
1kKvKdXcLvyHJTFuNxkdxm2tEnkG8hB1S
1QLT7GNBnKNrGnKi7HNnZTYSDbPvUaVoZs
12YwFZNmUoexnQAMKFvRJSSydxTZUZPWx8
1MDjCqjwnKmMWPxawpgN1UuDfbMUUgqnWw (You sent double $1 to someone)
1MDjCqjwnKmMWPxawpgN1UuDfbMUUgqnWw (<)
1LeXjaxMbugBveWusTRpbFtx13N5HamcE4
15j1jdJsMa4vR71gcZ6FCfYeAWJdPwpn7W
1MzdKiBn5qr8CS1cbts6TQquxsAo52yFKe
163vXnDXSc2hEKMrWHkERzBJWuuKJve5Nc

19Y4oNeGcdmBAfDXpJjPbiZ35PnZz57Ar2
1MtYZBtRw8XcnTgEY8qQphbddnwwFcoEXH
1JHoF2bak7KCST3SzeR7e1AwqDm4tiLJjt
1D6qsGhZqrRSKegqWT3e8TPR8gGczTxMLu
1AcKeSkKponv5qeZuEHvkocELf1Uo4ggCE
17GBpDP8yHTZcBJ9WmmRMjCmgiqacy5v3n
1fiWBi5u7oDzQrMNjeNLbmAvvHv545oy9
1NyBMbtqZLAmB3CSbjHzDHvedRJJJ7CupY
1N96tMSyeeANTRFApr3zA6ML3Qow1gZR9A
1ExEJE581mstkFPe5n5onKYHK9urFHCTTe (Someone got paid a bit more than $1)
1DuA2rvcJmgBLs9TVE14chN94mwsnW8xEe (And some people got paid less than $1)
1PQHK5ivZGyRf83TDZsvhnmecxKgCK4JXa
1BHD4CRjp1yLPhZDQY31A2vPRXUBKFt4JF
1FqQV1R3H8dftDupCjCoW7b6kSVxoBZBmh
1LveaSSVmvRvnbcwsvBdC8AvoqoLro5DG4 (Probably last person)
1GymPp2dave9ByFRcFAwkPX3bMeVyoZkPq
1AQ6fj7HUWkMWokyVFy3jv5XRALxg8VH2x (Definitely has to be last)

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.

19WmbY4nDcjAEv6wb5rcd5E6MutVMXBZzy = wallet
14J6ep326owXDpc1waViGJNB1onSFy9eou = purchase
I might've gone too far back possibly, but this was the farthest I think I could go without thinking that someone might've spent over 40,000 in one exchange for this class

c. (Bonus) Can you learn anything about where the send of the bitcoin is located geographically?

You can guess based off of transaction times as well as days when the person is more likely to be sending those transactions. At the same time, having traced most if not all of the professor to student transactions, you or your program seems to prefer Chicago, Northern England, the West Coast of Africa, Germany, Sweden, Florida, and some other odds and ends, which pretty much means that it is very, very difficult to figure out where they are located geographically. Besides that I probably couldn't really draw much else that I can think of.

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.

One could easily make a bitcoin program that simply takes any money that is put into it and puts it into the malicious person's own protected wallet. Besides that, along the same line of having a malicious program initially, you could steal the user's wallet words, passwords, private keys, etc. However along the lines of not getting caught, they'd probably be going for doing very, very minimal transactions that are only pennies at a time over the course of a long period of time, kind of like the transaction fees, but not to the knowledge of the user until it gets slightly factored into the new transaction fees every now and then.

(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?

I wouldn't say that I am too overly confident in the safety of my wallet I am currently using, nor would I say that I'd be too confident in any 3rd party software that managed my wallet. If I were to store all of my income in it, I would probably create the software myself or implement it on my own system.

Problem 4

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F

$16^{64} - 1 = 2^{256} - 1 =$
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

$16^{64} – 16^{8} - 1 = 2^{256} – 2^{32} - 1 =$
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF

….
$16^{64} – 16^{8} – 2^{9} – 2^{8} – 2^{7} – 2^{6} – 2^{4} – 1 =$
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F

Problem 5

I need to trust the code given to us and that there is no things on inside that are possibly hidden that will end up tainting the keys. I have to trust that the btcd is coming up with the correct hashes and keys. I have to trust that the math is correct using the elliptic curve secp256k1. I have to trust that the way in which I am launching the code will not be compromised by either being watched by someone else, recorded at any time, or by just generally being not safely used.

Problem 6

```
func generateVanityAddress(pattern string) (*btcec.PublicKey, *btcec.PrivateKey){
    for {
        priv, err := btcec.NewPrivateKey(btcec.S256())
        addr := generateAddr(priv.PubKey())
        matched, err := regexp.MatchString(pattern, addr.String())
        if (matched){
            return priv.PubKey(), priv
        }
        if err != nil {
            log.Fatal(err)
        }
    }
}
```

Problem 7

Vanity Address:
1BiXQ1UvxC2E2g4233x3tVBDEfvLviZPR3
Public Key:
[033c6bca1c228b98a423f339c66d99e23ed7580cd0d7295555a4e3a901e99bb062]

Problem 8

Technically the complexity is the same for both the previous address and the vanity address, but in reality, it is safer to have the vanity address because it is more difficult to make a similar looking address to the vanity address that isn't easily identified as fraud than it is to create a similar looking address to just a random address.

Problem 9

f3b1f0ff36c985ba69341cb99d90bb2bd269f25e6eee0fa328a2bb4ff89670a9

Problem 10

The transaction ID of the mistake:
4d46bde9cc7c714ccceaad2a1d97dd38ca2db15d2e929ac2440deb126e00ec43
I messed up and accidentally sent it back to my change address from when I sent to my vanity address.
So I sent myself a bit more (.2 USD) to my vanity address:
c74cec8d507baf63b9ca09b6b145c47a6cc27985a6872b64b07307b261e0ab2c

Problem 11

And after I edited my code to allow different sending amounts I sent about half (.1 USD) to a fellow student in class at:
112Kh8RHajxsj2yqdvrdcq5L4bLm8xymY
From my vanity address of:
1BiXQ1UvxC2E2g4233x3tVBDEfvLviZPR3
With public key:
033c6bca1c228b98a423f339c66d99e23ed7580cd0d7295555a4e3a901e99bb062
Previous transaction ID:
c74cec8d507baf63b9ca09b6b145c47a6cc27985a6872b64b07307b261e0ab2c
Which gave me a new transaction ID of:
dd0193146ea152fa26f1de59266559e3847cbdcb7f1a564758d090057f18a36d
Which gave about .1 USD to the student (for Problem 10) and also sent only part of what I had available through the modified spend.go program (for Problem 11)

spend.go

```
/*
** 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 to send Bitcoin to.")

// 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    uint64
}

// 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(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(err)
    }

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

    args := requiredArgs{
        txid:      txid,
        vout:      uint32(*vout),
        toAddress: addr,
        privKey:   key,
        amount:           uint64(*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) (*wire.TxOut,
*wire.OutPoint) {
        blkChnTxOut := rawtx.Outputs[vout]

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

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

        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)

        // 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(int64(reqArgs.amount), 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) }
```

keypair.go

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

// Every file in go is a part of some package. Since we want to create a program // from
this file we use 'package main'
package main

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

// generateKeyPair creates a key pair based on the eliptic curve // secp256k1. The key
pair returned by this function is two points // on this curve. Bitcoin requires that the
public and private keys // used to generate signatures are generated using this curve.

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

        // Generate a private key, use the curve secpc256k1 and kill the program on
        // any errors
        priv, err := btcec.NewPrivateKey(btcec.S256())
        if err != nil {
                // There was an error. Log it and bail out
                log.Fatal(err)
        }

        return priv.PubKey(), priv
}

func generateVanityAddress(pattern string) (*btcec.PublicKey, *btcec.PrivateKey){
        for {
                priv, err := btcec.NewPrivateKey(btcec.S256())
                addr := generateAddr(priv.PubKey())
                matched, err := regexp.MatchString(pattern, addr.String())
                if (matched){
```

```
                **return priv.PubKey(), priv**
            **}**
            **if err != nil {**
                **log.Fatal(err)**
            **}**
        **}**
**}**
```

// generateAddr computes the associated bitcon address from the provided // public key. We compute ripemd160(sha256(b)) of the pubkey and then // shimmy the hashed bytes into btcsuite's AddressPubKeyHash type func generateAddr(pub *btcec.PublicKey) *btcutil.AddressPubKeyHash {

```
        net := &chaincfg.MainNetParams

        // Serialize the public key into bytes and then run ripemd160(sha256(b)) on it
        b := btcutil.Hash160(pub.SerializeCompressed())

        // Convert the hashed public key into the btcsuite type so that the library
        // will handle the base58 encoding when we call addr.String()
        addr, err := btcutil.NewAddressPubKeyHash(b, net)
        if err != nil {
                log.Fatal(err)
        }

        return addr
}

func main() {

        // In order to recieve coins we must generate a public/private key pair.
        pub, priv := generateKeyPair()

        // To use this address we must store our private key somewhere. Everything
        // else can be recovered from the private key.
        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()))

        addr := generateAddr(pub)

        // Output the bitcoin address derived from the public key

        fmt.Printf("This is the associated Bitcoin address:\t[%s]\n", addr.String())
        pub2, priv2 := generateVanityAddress("^1BiX")
        fmt.Printf("This is a private key in hex:\t[%s]\n",
                hex.EncodeToString(priv2.Serialize()))
        fmt.Printf("This is a public key in hex:\t[%s]\n",
                hex.EncodeToString(pub2.SerializeCompressed()))
        vAddr := generateAddr(pub2)
        fmt.Printf("This is the associated Bitcoin address:\t[%s]\n", vAddr.String())
}
```