Jacob Freck
jef6mt
9-15-15

Problem Set 1

Problem 1.

a.

Transaction ID: 539518e2dbf578f70a846eb63f920ccb10dc3f7e41321a68fa1c6e9324f10a9c

b.

Fee: 0.0001BTC or ~.02 cents USD

c.

Total Ouput of block: 7,531.39109632 BTC

d.

It should have taken about 10 minutes per confirmation, and about 30 minutes for 3 confirmations. Bitcoin is designed to add a block to the blockchain every 10 minutes, with each block confirming new transactions and building upon (or re-confirming) the previous blocks' transactions. So 3 confirmations would be around 30 minutes.

According to blockchain.info, however, the payment was received 6 minutes before inclusion in it's first block, so 36 minutes may be a more accurate estimate.

Problem 2.

a.

1. 1QLT7GNBnKNrGnKi7HNnZTYSDbPvUaVoZs
2. 1kKvKdXcLvyHJTFuNxkdxm2tEnkG8hB1S
3. 1GMrGqvF8FwbgCSShCdzijcSUHzmNR1VCz

b.

Transaction ID: 66809ee9b41310b53e3fc94cbff8fe7c870d6df4ca7561614137311f651ed9a8

Time: 2015-08-26 20:45:29

This is probably where the bitcoin was original purchased. Because it's a nice, even amount and the subsequent transactions were all deposits of this .2 BTC coin to different addresses of the same amount I received.

c.

On blockchain.info, there is an ip address that the transaction was relayed by. This IP, 54.212.89.254, address belongs to Amazon Technologies. This IP address seems to be located in Portland, Oregon. Thus, one can surmise that the transaction was probably sent from the US, and possibly even the west coast. However, since Bitcoin does not store any information as to the sender's specific IP or any other hints at the sender's location, this is just a guess. A savvy enough user could hide their location, or falsify the IP they wish to use, or the like.

Problem 3.

a.

A malicious wallet application might steal user's bitcoin without detection by implementing a fee of their own similar to the mining fee. Stealing tiny amounts of coins, across huge user bases could be an enormous source of value, while only stealing minuscule amounts from each user at a time. This would likely avoid detection from unsuspecting users, but would definitely leave a trail, as each of these phony transactions would be stored on the blockchain.

A malicious developer might also save each private key each user creates with their application. This developer need not steal a user's bitcoin right away, but might be better served by stockpiling these private keys, waiting for the application to grow in user base. Then, the developer could steal all the bitcoin from all the private

keys stored at once. This would not raise any alarms while keys are being collected as no value appears to be lost on the users end, and the developer would only have to launder or cash out the coins in one lump, rather than micro amounts each time.

b.

I am fairly confident that my money is safe, although I realize this is naive. I trust the wallet application because I read online from multiple sources that it was a trustworthy program. However, this is only suitable really for small stores of money. If my wallet were to hold any significant value, I would not feel comfortable using a certain wallet implementation without first thoroughly searching online for potential issues or security flaws. Also, I would look at the source code of the application itself to see if I could identify any problems or suspicious activity.

Really, however, at some point, I would just have to trust that since other people are engaging in transactions without major issue, I can do the same. This may feel uncomfortable, but there is no fundamental difference between trusting bitcoin or trusting the bank. At some point, you have to rely on others to store your money, whether that be in crypto currency, a bank account, or cash (trust gov.).

Problem 4.

```
package main

import (
	"math/big"
	"fmt"
	"encoding/hex"
)

func main() {
	correctval := powInt(256)
	correctval = correctval.Sub(correctval, powInt(32))
	correctval = correctval.Sub(correctval, powInt(9))
	correctval = correctval.Sub(correctval, powInt(8))
	correctval = correctval.Sub(correctval, powInt(7))
	correctval = correctval.Sub(correctval, powInt(6))
	correctval = correctval.Sub(correctval, powInt(4))
	correctval = correctval.Sub(correctval, big.NewInt(1))
	fmt.Println(hex.EncodeToString(correctval.Bytes()))
}

func powInt(pow int) (*big.Int){
	i := 0
	val := big.NewInt(1)
	two := big.NewInt(2)
	for i < pow {
		temp := new(big.Int)
		temp.Mul(val, two)
		val = temp
		i += 1;
	}
	return val
}
```

This code produces the same modulus as the one used as secp256k1.P in btec.go.

Problem 5.

Things you need to trust in keypair.go:

1. good randomness

   You trust that keypair.go uses a source of randomness that is perfectly uniform and evenly distributed across all possible values so as to preserve all entropy.

2. You need to go look at the functions in all of the imported libraries.

   ie. you can't trust that the btec NewPrivateKey() function does anything cryptographically secure unless you go and inspect it. For all we know from keypair.go, it generates a key pair and publishes it to a website for everyone to see.

3. You must trust that discrete logs are hard

   since this is the underlying assumption in most of modern cryptography, you must trust that it is right even though it is not proven.

4. you must trust that no other program has access to the output of these functions.

The output is shown in plaintext on the console, and presumably stored somewhere in memory at the time keypair.go is run, so you must trust that no other program or malicious entity is "watching."

5. You must trust the security of the go programming language.

Plenty of programming languages have had implementation security flaws, go is probably not an exception to this. So you must hope that there is no security flaw in keypair.go exploitable because of programming language security holes. Perhaps, for example, the go language standard for string has some exploitable flaw that allows an attacker access to our private key.

Problem 6.

```
func generateVanityAddress(pattern string) (*btcec.PublicKey, *btcec.PrivateKey) {
	pub, priv := generateKeyPair()
	addr := generateAddr(pub)
	for {
		matched, err:= regexp.MatchString(pattern, addr.String())
		fmt.Println(matched, err)
		if matched {
			return pub, priv
		}
		pub, priv = generateKeyPair()
		addr = generateAddr(pub)
	}
	return pub, priv
}
```

Problem 7.

Vanity Address: 1VANC2T3xEHa3atU42iFP2y1xbTEUFFtf

starts with VAN (for vanity)

Since the algorithm simply generates a keypair and checks if the address matches the regex pattern, the run time complexity will be exponential. Since there are 58 possible values for each character, it would have a runt time of 58^n where n is the number of characters in the vanity address (assuming all characters are equally as likely).

Problem 8.

Technically, it is neither more secure nor less secure than the first address, as it is simply a hash like any other. There is no security benefit from a technical standpoint at all. However, in practice, having a vanity address may allow you to better recognize and identify your bitcoin address from other addresses. You might be less likely to mistake your address if it has some easily identifiable information, such as english text strings in it.

Conversely, however, you may be at a larger risk of mistaking an address that has matching vanity portions. For example, if you vanity address starts with your first name, likely you will verify this portion of the address without considering the rest of it very closely (or at all). Thus, if it were replaced with another address with the same vanity portion, you may be at a larger risk of misidentifying that imposter address as your own.

Still, however, from a technical standpoint, an address composed entirely of grammatically correct english text is no less or more secure than an address that appears random. Only human error can account for any added/lost security.

Problem 9.

Transaction id: 698ef4718d7bcb20570cd03011d840953c731ceb7db3a7b9c919ac8bea77dfeb

Problem 10.

Output:

```
$ go run spend.go -privkey "6b…9" -toaddress "1GMrGqvF8FwbgCSShCdzijcSUHzmNR1VCz" -txid
"698ef4718d7bcb20570cd03011d840953c731ceb7db3a7b9c919ac8bea77dfeb" -vout 0
Here is your raw bitcoin transaction:
0100000001ebdf77ea8bac19c9b9a7b37deb1c733c9540d81130d00c5720cb7b8d71f48e69000000006a4730440220212cda9422dd1ed216ad17fdc7
20a7bffd4253d46ef9959ae05db52f38e4f288022046329714dd6f6765594e53ea7477aef1616045628690cde23eaaee4d2eb55e57012103ac252a08
1ec7ffc04dc52c34de30d08137cfc1fe3efcbd75730fa1598c9456e7ffffffff01905f010000000001976a914a87b564feba7b0b2b1007a416f6497
d9d3af947088ac00000000
Sending transaction to: https://insight.bitpay.com/api/tx/send
The sending api responded with:
{"txid":"5d163abe0ceea96f2c18239cb703241001958adde95b6fa35c391bbb86a3535b"}
```

Problem 11.

Code:

```
/*
** spend.go (modified)
**
** 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 in Satoshi to send")

// 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    int64
}

// 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), the index into the outputs of that tx that fund your\n" +
                  "address, and the amount to send. 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:        int64(*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)

	//check if amount > oltTxOut.Value
	if reqArgs.amount > oldTxOut.Value {
		fmt.Println("amount exceeds funds")
		return
	}
	// 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(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)
}
```