Problem Set 1

Luke D. Gessler (ldg3fa)

CS 4501-001

September 9, 2015

Problem 1

a. The transaction ID was `84673335c59246ca20402e7ce0e49d4524e66c9f71a84786f9dd15917ca9c29e`

b. According to blockchain.info, the fee was 0.0001 BTC

c. The output total was 24,496.2120844 BTC

d. This block, 372400, is timestamped 17:13:02. Three confirmations later, at 372403, the time was 17:41:47. This is a difference of 28m45s. This answer is not surprising, as we know that block hardness is adjusted dynamically so that a block is solved, on average, every 10m. So three confirmations came a little earlier than expected, but quite close to the expected 30m.

Problem 2

a. Working back from "my transaction", I look at where the sum originated and see that the previous transaction resulted in 3.14159 mBTC going to `1PQHK5ivZGyRf83TDZsvhnmecxKgCK4JXa`. I do the same again and see that 3.14159mBTC also went to `1DuA2rvcJmgBLs9TVE14chN94mwsnW8xEe`. Now working forward from my transaction, I see that from the remainder another 3.14159 mBTC went to `1FqQV1R3H8dftDupCjCoW7b6kSVxoBZBmh`.

   That the same amount was used in succession and that the amount was the first few digits of π makes it quite likely these addresses also belong to students in 4501-001 and that a 4501-001 course instructor was the one distributing these coins.

b. I pass back through all the transactions in late August that sent 0.004501 BTC (or a similar conspicuous number that amounts to around 1 USD) to (presumably) students in CS 4501-001. This leads me to `19WmbY4nDcjAEv6wb5rcd5E6MutVMXBZzy`, where the 0.2 BTC that was distributed to students originated. This transaction also sent a not-so-pretty number of BTC, 0.40180117 BTC to another address.

   Following the "remainder" coins leads to no discernible pattern of transactions like we saw with the 0.2 BTC. At the time of the transaction (26 Aug), registration in 4501-001 had stabilized in the 30's. Note also the below, demonstrating that 0.2 BTC was just enough coin to safely cover everyone in the class[1].

$$0.2 \text{ BTC} \times \frac{1 \text{ student}}{4.501 \text{ mBTC}} \approx 44 \text{ students}$$

   All of this is consistent with the claim that the total sum (0.60180117 BTC) in transaction `66809ee9b41310b53e3fc94cbff8fe7c870d6df4ca7561614137311f651ed9a8` was controlled by someone else—probably a coin exchange—until Prof. Evans purchased an even number (0.2 BTC) to distribute to his 4501 students. The remainder was sent to another address controlled by the coin exchange.

[1] with some room for negligent students who spend all their BTC and need more

Problem 3

a. Method 1. **Motivation.** It might be a reasonable assumption that most people never empty their wallet entirely. It's encouraging that this is the same assumption that real-world banks operate on: banks invest a portion of the money that people entrust them with. Just enough money is kept liquid to meet demand for customers who wish to withdraw. Time has shown this is a reasonable assumption, except in extraordinary circumstances[2].

Explanation. If this assumption turns out to be true of bitcoin wallets as well, we could transfer a percentage $x \in [0, 1]$ of our clients' funds to our own wallets (and not tell them), where x would probably be determined for each client by an algorithm after monitoring their usage for a few weeks.

So if we took this amount, our client would believe she has W_{supposed} bitcoins but would actually have $W_{\text{supposed}} = W_{\text{actual}} + x \times W_{\text{supposed}}$, and we would have $W_{\text{stolen}} = x \times W_{\text{supposed}}$.

But sometimes someone might indeed empty their entire wallet. In this case, we would need to include an amount equal to W_{stolen} in the transaction to avoid detection. This is an acceptable outcome for the malicious developer, as long as two conditions hold:

i. At any given time, it is highly unlikely that many users would empty their wallets.
ii. Our users collectively have enough bitcoin to meet demand in transactions that require more than a given user's W_{actual}.

Now, as long as the developer is able to determine a safe $W_{\text{buffer}} < W_{\text{stolen}}$ to handle the case in which the user withdraws more than in W_{actual}, the developer can withdraw $W_{\text{stolen}} - W_{\text{buffer}}$ for her own use.

Outcome. The user, as long as her only way of knowing about her addresses is our client, would never detect any of this malicious work as long as we make sure the GUI properly displays what their balance is supposed to be.

The biggest weakness is that a technically skilled observer would take a look at the blockchain and deduce what's going on. (Obviously, we would not open-source our code.)

Method 2. Bundle a bitcoin mining daemon with the wallet. Add bells and whistles as necessary (suspend mining during heavy use, throttle CPU/GPU to avoid conspicuously loud fans/hot hardware, hide/disguise the process, etc.). This is not stealing in the sense that we're taking our users' coins, but we are stealing their energy and computing power.

b. **My current wallet.** Because multiple blockchain sites (accessed on different devices on different networks) have confirmed that I have 3.14159 mBTC in my wallet, and because I know my address belongs to me since I created it and hold its wallet words, I'm reasonably confident I do in fact control the bitcoins.

If I wanted to be more sure, I would take my wallet words and use a different (or my own) client on different hardware on a different network to restore my wallet from my wallet words.

One remaining possibility is that my client also communicated my keys, etc. to an evil person, but I continue to use MultiBit on the assumption that because many people use it and nobody has had this happen to them it's probably not happening.

Paranoid measures. If I were more paranoid, I might...

1. regularly move my bitcoins across different clients I have on different machines on different networks. The cost of this is negligible since the fee is flat, not a percentage of the transaction, and it would be robust against Method 1 explained above in a.
2. try restoring my wallet from my wallet words on different devices after creation to verify authenticity.

[2] E.g. the stock market crash of October 1929, when banks did not have enough liquid assets on hand to meet demand for customers who wished to liquidate their accounts.

3. avoid websites to analyze the blockchain and store a local copy instead.
4. look into sophisticated tools that make the blockchain human-intelligible, with the hope that I would be able to see all my coins and ensure that they are at addresses that belong to me.
5. not use a client at all and use calls to the bitcoin API I code myself, or at least a very thin wrapper around it. This avoids using a binary someone else told me is trustworthy.
6. consider only using a paper wallet I keep in a safe. This would mean not storing my private keys electronically at all, but rather keeping it in e.g. a QR code on paper, taking it out and keying it into something like the CLI for `spend.go` only when I need to move funds.

Problem 4

This probably wasn't the best way to do it, but it works:

```
func main() {
  p1, _ := big.NewInt(0).SetString("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F", 16)
  fmt.Printf("%v\n", p1)

  two := big.NewInt(2)
  p2 := big.NewInt(0).Exp(two, big.NewInt(256),nil)
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(32),nil))
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(9),nil))
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(8),nil))
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(7),nil))
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(6),nil))
  p2 = p2.Sub(p2, big.NewInt(0).Exp(two, big.NewInt(4),nil))
  p2 = p2.Sub(p2, big.NewInt(1))

  fmt.Printf("%v\n", p2)
}
```

First we read in the hex string in btcec's code and print it out.
Then we compute it from the formula $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.
The two results are the same.

Problem 5

I need to trust that:

1. my machine is not compromised (e.g. through a backdoor). If it were, it would let someone tamper with my information (like the source code for `keypair.go`, which would let them e.g. display a "random" private key that they actually own) or transmit my information (like my private key).

2. the discrete log problem is hard. This is foundational for the cryptography used in `btcec`. My private key could be guessed from my public key if this turned out to be false.

3. `btcec`'s implementation of the API calls used used in `keypair.go` is correct. This is probably the case, as it's used by thousands of people without incident.

4. I have a correct copy of the `btcec` library. This depends on how I'm connecting to GitHub: a dedicated adversary might compromise my ISP's hardware and make me see a different version of `btcec` depending on the nature of my connection with GitHub.

Problem 6

The function has been defined in my `keypair.go` file.

Problem 7

My address is `1Luke788hdrUcMqdb2sUdtuzcYqozXgh4L`

Problem 8

Ignoring human factors, a vanity address is just as difficult to break as any other address. It unique quality—having a substring in its base 58 encoding that incidentally has special meaning to humans—does not change anything about its security. But when we take practical concerns into account, it's unclear whether a vanity address ultimately becomes more or less secure.

Imagine one use case of an address that involves many people sending coins to it. If a meaningful substring is in the address (e.g. PETA in 1PETAQ2TwX2yJSDZYrag2QWLio5ZK6ShEx) then it can help the user gain confidence it is the right address. But this cuts both ways. If we encourage users to focus on only this one substring, then we leave the door open for an adversary with a slightly different address, e.g. 1PETAQ2TwX2yJSDZYraq2QWLio5ZK6ShEx, to masquerade as PETA and trick users into being less cautious about suspicious addresses than they might be and thus giving funds to the wrong organization.

Problem 9

Transaction viewable at https://blockchain.info/tx/7b04780b0b875cbdd987aab21db79f204e1fb5be993f512e28d3b34c148

Problem 10

Transaction viewable at https://blockchain.info/tx/93c22c74627b7e69d0ac11148732d245039e12e6d108bb244402e5745d

This is one of Sam Prestwood's addresses.

Problem 11

The -amount flag has been implemented. The remainder is redirected back to the sending address. For an example transaction I made using this function, see https://blockchain.info/tx/cb9e8ec8ad02d0edd7b7d9abb85b2312304ff
in which I start with .5 mBTC transaction, use .1 mBTC as transaction fee, give .2 mBTC to my other address, and have .2 mBTC (the remainder) given back to the address.
See `keypair.go` for full changes.

Problem 12

I don't have time to do actually do this, but this is how I would try to double-spend:

1. Rent a VPS in India
2. Install Go and copy over the script used earlier to spend: `go run spend.go -privkey "77da..f9ddb" -toaddress "1MV8oVUW..nEm4UXM" -txid "e1..d0780" -vout 0`
3. Modify this copy of `spend.go` to broadcast to an Indian bitcoin API (rather than the BitPay Insight API)
4. Schedule both the local machine and the Indian machine to execute the script at x milliseconds after the epoch.

What should happen is that the two will broadcast a transaction with the same source transaction at the same time to different nodes in the bitcoin network. The two nodes should be far away enough from each other that they will not hear about each other's transactions until a few hundred milliseconds (at least) have passed. This means the network will have to have some kind of way of mediating these conflicts.