Digital Cash as a Web Service

Shawn O'Neil – Fall 07 Cryptography Project

December 4, 2007

1 Project Overview and Goals

The goal of the project was to build an easy to use digital cash "bank," implemented as a website which allows users to log on using a password protected account to deposit and withdraw cash. Cash which is withdrawn is signed by the bank, and the cash and signature are embedded into a jpeg picture of the user's choice using a steganography library.

Having the cash embedded in this way makes transferring cash to other users a painless operation; emailing the image will work as well as any other method of transfer. Users who then receive the cash can then deposit the cash to their account by uploading the picture using their account website.

The digital cash protocol used is Protocol 2 found on page 140 of [1]. This protocol ensures anonymity of cash: If Alice withdraws money and later spends it or redeposits it, the bank is unable to link that piece of cash with Alice. Also, this protocol protects the bank from double depositing: each piece of cash is issued a random serial number, and the bank checks that a piece of cash has not been deposited already before accepting it.

On the other hand, this particular protocol doesn't protect users from the double spending problem in a strong sense. If Alice has a piece of cash, she can spend it at Bob's Bookstore and later at Lisa's Liquor Store, and whichever tries to deposit the cash second will have it refused by the bank. To try and remedy this to some extent, anyone (even those without an account) will be able to validate a piece of cash with the bank to ensure that it

1. Is actual cash that has been signed by the bank,
2. Is worth the value they think it is, and
3. Has not yet been deposited.

Thus, merchants or anyone receiving cash are advised to immediately verify the cash and deposit it. (See the later section on verifying cash for other advice users should take to ensure privacy in this regard.)

1.1 The Protocol

1. Alice prepares 100 anonymous money orders. On each she includes a different random uniqueness string, such that the chance of this string being reused is small.
2. Alice blinds each money order, and sends them all to Bob, the bank.
3. Bob unblinds 99 of the money orders at random (with Alice's help), and verifies that all the amounts match.
4. Bob removes the appropriate amount of money from Alice's account, signs the remaining still-blinded money order, and returns it to Alice.
5. Alice unblinds the money order, and spends it with a merchant.
6. The merchant checks for Bob's signature to make sure the cash is valid. (He can also ask the bank, anonymously if desired, to check that the cash has not already been deposited.)
7. The merchant takes the cash to Bob.
8. Bob verifies the signature, and checks to make sure the cash has not been previously deposited by looking for the uniqueness string in his

database. If not, he adds the amount indicated to the merchants account, and records the uniqueness string for future checking.

2 Implementation Details

2.1 Blind RSA Signatures

The digital cash protocol requires the use of blind signatures. While many commercial cryptographic tools support signatures, most that I am aware of first hash the message to be signed and then sign the hash. This is not acceptable for a blind signature scheme, we must sign the original message (which is multiplied by a blinding factor raised to the public exponent), and we assume that the message is sufficiently short (less than n when represented numerically).

So, I implemented a simple RSA encryption and signing library using the Ruby programming language. Using Ruby helped to speed the development time for this, as large number support is Ruby's default behavior, and converting between strings, numbers, and chars is fairly straightforward. Speed so far hasn't been an issue, I can compute 100 signatures in approximately 20 seconds on the 666 Mhz PIII cpu the code is running on. As we shall see later, the most time consuming step of the whole protocol is getting ahold of random bits for the cash serial numbers.

512 bit primes primes for the main server keys were generated using `gpg --gen-prime`. Standard techniques for fast exponentiation and inverses modulo a prime were used as described in [2].

The RSA test vectors I found were useful only for implementations conforming to full RSA standards, including padding before encryption and hashing before signing. As such, I wrote my own small test set, which generates a batch of cash (see below) and verifies that each piece can be blinded, signed, unblinded, and verified successfully. This test can be run by calling `bob.rb testvectors` on the command line. To (attempt) to verify that signatures can't be used for unauthorized messages, there is also a version of this test which adds a small random amount to the value of each piece of cash, and attempts to run the verification using this modified cash. The command for this version is `bob.rb testvectors cheat`.

ASCII strings are converted to numerical form by taking the decimal number of each character and concatenating them all together. Thus, because the decimal code of the character b is 142 and the code for o is 157, "bob" is equivalent to 142157142. In the case of leading 0's, which are cut off in the numerical representation, we can simply add an appropriate number of leading 0's back on for conversion back to ASCII, noting that the number of digits in the numerical representation should be a multiple of 3.

2.2 Generating Cash

The lower level Ruby scripts also handle the generation of cash in batches of 100 for Alice. When Alice requests a batch of cash (with value, say, 37) for use in the protocol (which is handled at a higher php level, see below), 100 entries of the form "37:1fEkXTL53VCDC45U" are returned. The number before the : is the value of the cash, the other 16 characters make up the unique serial number. Each character is drawn randomly from uppercase letters, lowercase letters, and numbers.

In creating such large amounts of random numbers, I quickly found that the machine this runs on lacks an appropriate source of entropy. Fortunately, `https://www.random.org` provides an alternative source for such random strings, ultimately generated from atmospheric noise. The connection to the site runs over SSL/TLS (using AES-256) for added security.

This portion of the protocol takes the longest, with a noticeable wait on the order of a couple of seconds occurring while the user is waiting for their signed cash to be returned.

2.3 Code Access Layout

The largest flaw in this project is the fact that while Alice and Bob are separate entities in real life, the protocol and actions taken by both parties are implemented as code which runs on a single server. However, to keep to the spirit of digital cash, we separate Alice (the user) and Bob (the bank) at a code/object level.

(A more sophisticated implementation might use a client side application for requesting cash, perhaps something as integrated as a java applet. This leaves open the question of shifting the trust from the bank to the writer of the applet or a community of auditors, as not all users can be expected to perform their own security audits.)

At the lower RSA signature level, Bob has his own set of Ruby scripts for various cryptographic functions, including signing. The public and private keys are stored in simple text files, owned and readable only by the `www-data` user. At a higher level, there is a Bob php object which handles requests from a separate Alice php object (they both have pointers to each other so that they can run function calls). Also, logically part of Bob's functionality is a database interface object which only the Bob object has access to, and of course the database itself, which only the database interface object has access to.

Everything else in the system can be regarded as "Alice." This includes the user interface, the Alice php object, the steganographic library, and Alice's lower level Ruby code, which can be used for generating cash, blinding, unblinding, verifying signatures, and so on.

2.4 Password Management

User password checking is handled by Bob, as username/hashed password pairs are stored in the database. The hashed password is computed as `crypt(md5(<password>),md5(<username>))`. While the php documentation is a little unclear, crypt should be using MD5 hashing, taking the first 8 characters of the second argument as a salt.

For security, the server supports SSL/TLS using AES-256 for session encryption of all communication between the client's browser and the server. The certificate is a "dummy" test certificate.

2.5 Steganography

A php class known as `stegger` is used to embed the cash and the signature in a jpeg picture of the user's choice. The returned image is in png format. This class wasn't quite as general as my needs required, so some small changes were necessary to make it interface with the rest of the code properly.

Fortunately, there are no cryptographic requirements on the security of the steganography library; images are merely containers to facilitate transfer of digital cash.

3 Protocol Overviews

3.1 Withdrawing Cash

After a user Alice logs in to the website, she is presented with her current balance, and has the option of withdrawing some cash. She selects a jpeg picture and enters and amount to withdraw.

At this point, the Alice's portion of the code is asked to generate a batch of cash with random serial numbers, which are all converted to a numerical form and blinded with random blinding factors between 1 and 2^{28} (also retrieved from `https://www.random.org`). These are all sent to Bob to initiate the signing protocol.

When Bob receives these 100 blinded pieces of cash, he picks a random number r between 1 and 100 and returns it to Alice. In effect, Bob promises to sign the blinded message r if Alice can prove to him that the other 99 are well formed.

Alice responds by sending back to Bob the other 99 blinding factors, as well as ASCII versions of the cash itself. Bob is then able to check that 1) all values on the cash are equivalent, 2) that Alice has that much cash available in her account, and 3) that the blinded versions of the cash match the blinded cash he received in the previous step.

If those conditions are met, Bob first removes the appropriate amount of money from Alice's account (so that Alice cannot first get cash and then somehow interrupt the deduction process), signs the blinded cash he promised to, and returns the signature to Alice.

Alice, finally, embeds the original ASCII version of the cash Bob has signed along with the signature returned into the jpeg file, which is returned to the user.

3.2 Verifying Cash

As mentioned in Section 1, because the protocol prevents double depositing, but not double spending, a mechanism is provided for users to verify that a piece of cash they have is properly signed, undeposited, and of the correct value.

One thing to note is that, in order to verify that a piece of cash has not been deposited, the Alice portion of the code needs to *ask Bob* to check this, as Bob is the only one with access to the database. However, because verifying cash isn't linked to the user's identity, an interface is provided for doing this so that any person, even those without an account, can verify cash without logging in. If a user is logged in a verifies some cash, the bank will know that a particular user had a particular piece of cash at a particular time. It is suggested that users use this functionality. From a public coffeehouse. In Belgium.

To verify some cash, the user uploads a digital cash bearing png file, from which the Alice portion of the code extracts the cash and signature. The signature is verified, and Bob is called upon to check if the cash is undeposited. The value, serial number, deposited status, and signature validity are returned.

(Alternatively, the deposited cash database itself could be made publicly readable. Aside from being practically semantically the same as the solution provided, this exposes the entire depositing history, which would be an undesirable trait.)

3.3 Depositing Cash

The protocol for depositing cash is extremely simple. The cash bearing png file is uploaded, the cash and signature are stripped out and given to Bob. Bob engages in the verification protocol with himself, and if all is well deposits the amount of cash into the user's account.

4 Security Analysis

As mentioned, the weakest part of this system is the trust which is required of the server to faithfully run the Alice/Bob protocol using separate code bases.

As a separate concern, the user must also trust that the source of almost all the random strings and numbers, `random.org`, is not colluding with the bank to trace pieces of cash.

For protocol parameters, the size of the RSA modulus used should make it very difficult for the signature algorithm to be broken by cryptographic means. The space of serial numbers (10^{56}) is large enough that accidental serial duplicates is extremely small. Also, the space of blinding values (2^{28}) is large enough that it should be quite infeasible for the bank to unblind the cash being signed. Alice's chance of success at cheating is at most 1/100 by the cut and choose protocol.

Assuming the trust model described, the weakest point of the whole operation is most likely the server. While I certainly attempt to secure the system from outside attacks, these are certainly possible, given the large number of applications this computer runs.

5 Performance and Scalability

RSA based signing, blinding, and unblinding are all computationally expensive procedures. Methods do exist for speeding these operations somewhat, and any large scale application of digital cash will need to think about the load on the servers. Ideally, a more robust and efficient library could be found or produced for blind signatures.

In this application, the server is responsible for creating the cash serial numbers and random blinding factors, and so needs access to a large source of entropy. The solution used, an http request to a random string generator, would not scale well. Already there is a noticeable wait while cash is generated for the user.

On the other hand, in a commercial digital cash service, one would presume that each user would generate and blind the cash for themselves rather than let the server do the work. In this case, the "entropy load" on the server would be greatly reduced, as each user would only need to bring a small amount of entropy with them, so to speak.

6 Final Notes

The attached source code does not represent all code used for the project, but should represent the most important parts. This includes the cash withdrawal protocol, verification protocol, and depositing protocol. The files are presented in a "top down" order: first (some of) the UI, then Alice and Bob's php classes, then the Ruby interfaces for signatures/verification/cash generation, and finally Alice and Bob's cryptographic primitives libraries.

Currently, the project can be accessed live at `https://oneilsh.gotdns.org/bobcash`.

Bob is the name of the chipmunk who lives under our porch. Recently, there has been a second chipmunk around—we named her Alice.

References

[1] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[2] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications).* Chapman & Hall/CRC, November 2005.