Security Principles, SPR

21st – 25th January 2013

ASSIGNMENT

Mayur Pant

10th March, 2013

48 pages

Contents

1. Deducing security requirements by looking at system threats and vulnerabilities

Scenario summary

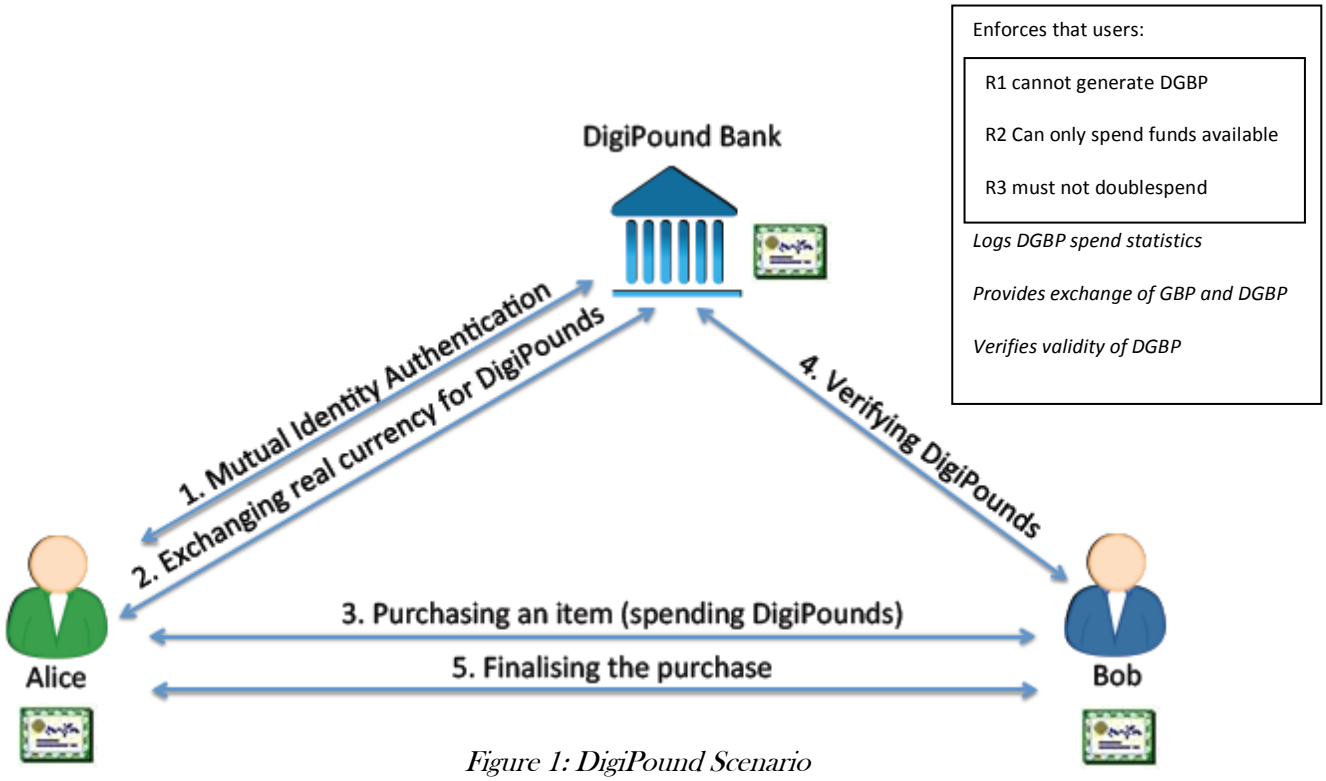
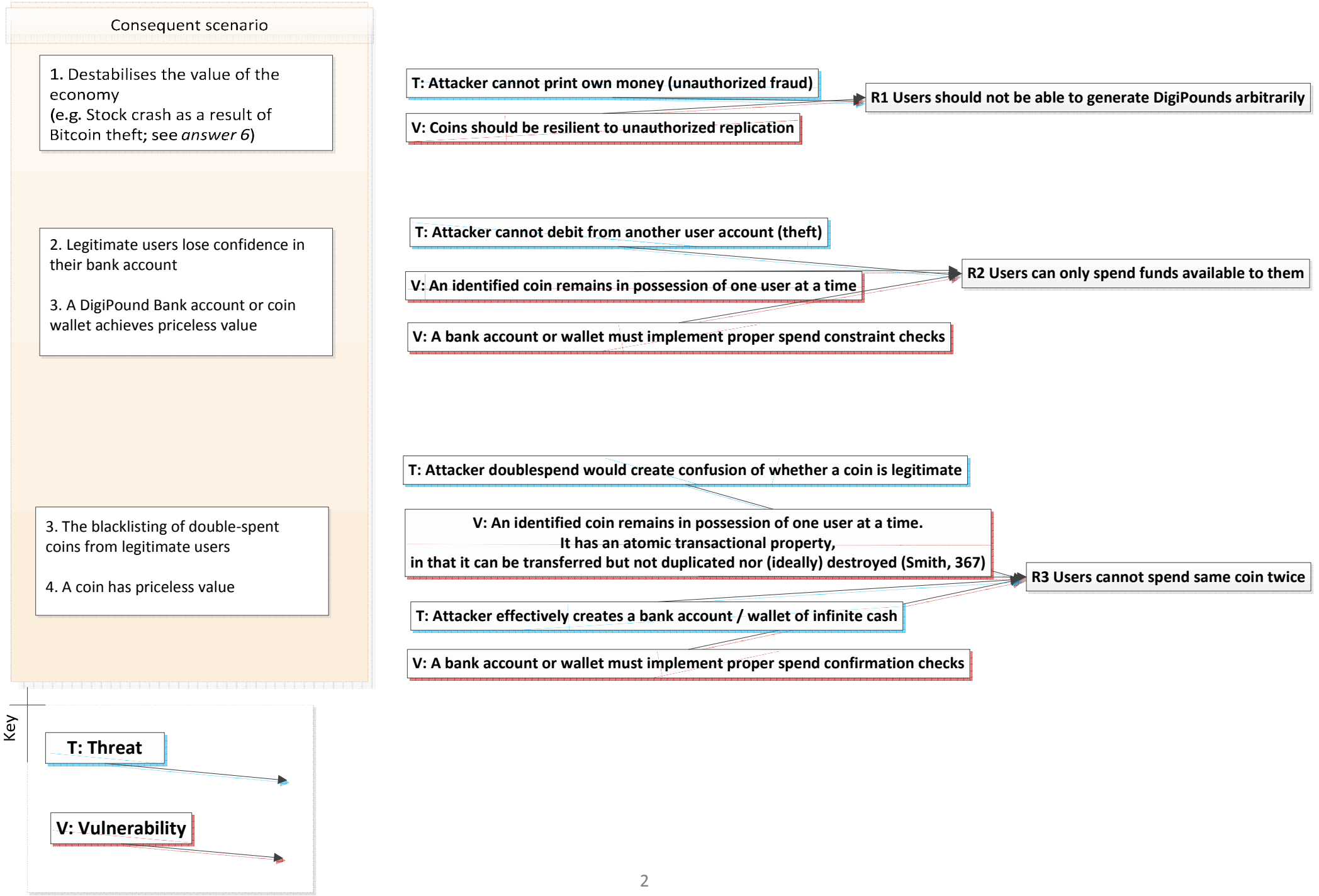


Figure 1: DigiPound Scenario

Exchanges

[1.] Mutual ID authentication: Alice<->DigiPoundBank
[2.] Alice-GBP->DigiPoundBank Alice<-DGBP-DigiPoundBank
[3.] Alice-purchaseRequest->Bob Alice<-signedPurchaseRequest-Bob (in stock) Alice-DGBP->Bob
[4.] Bob<DGBP_chck>DigiPoundBank ? [5.]
[5.] Bob-signedBill->Alice (DGBP verified)

Requirements, vulnerabilities and threats diagram



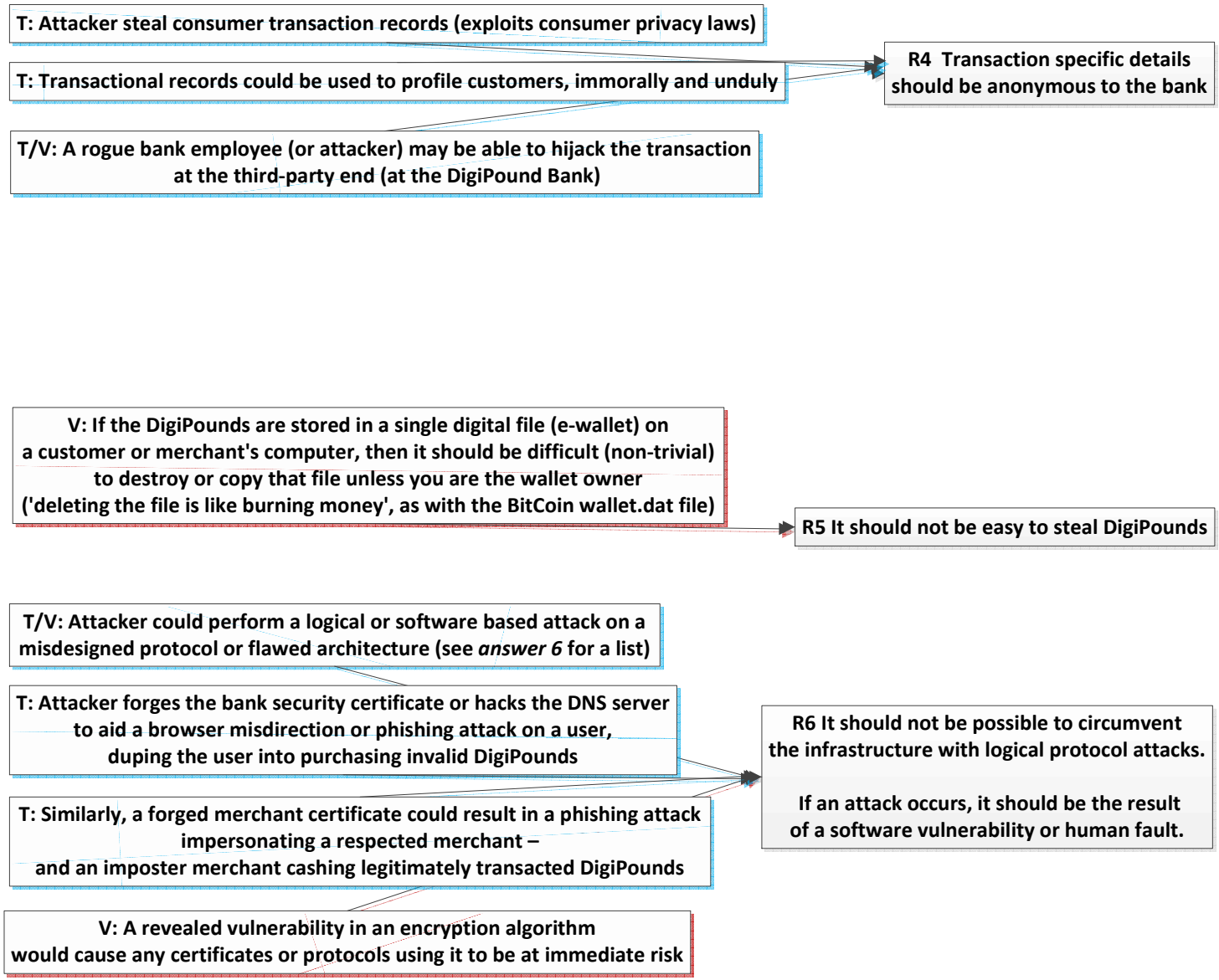
Consequent scenario

Any rogue merchant or employee could easily access a trove of confidential, valuable data, and use it or sell it to criminal organizations.

Similarly, we want Alice's banking details (that she used to purchase the coins) to be anonymous to the merchant; but in this scenario it is a given, as the purchased DigiPounds act as a medium (such that she never needs to hand her creditcard details to Bob). Divulging only the **necessary information** is always a vital business requirement.

A widespread virus steals or deletes from user DigiPound files.
Example shown in *answer 6*.

A merchant uses a flawed protocol with AES encryption, and a certificate with a cracked hashing scheme (MD5). The attacker performs a chess grandmaster attack when the customer sends the merchant Digicash, and impersonates the merchant's certificate to cash DigiPounds from the DigiPound Bank (easy if the merchant relies on CSL). Further illustration of concept at the start of *answer 6*.



Illustrating points for risks, vulnerabilities and threats

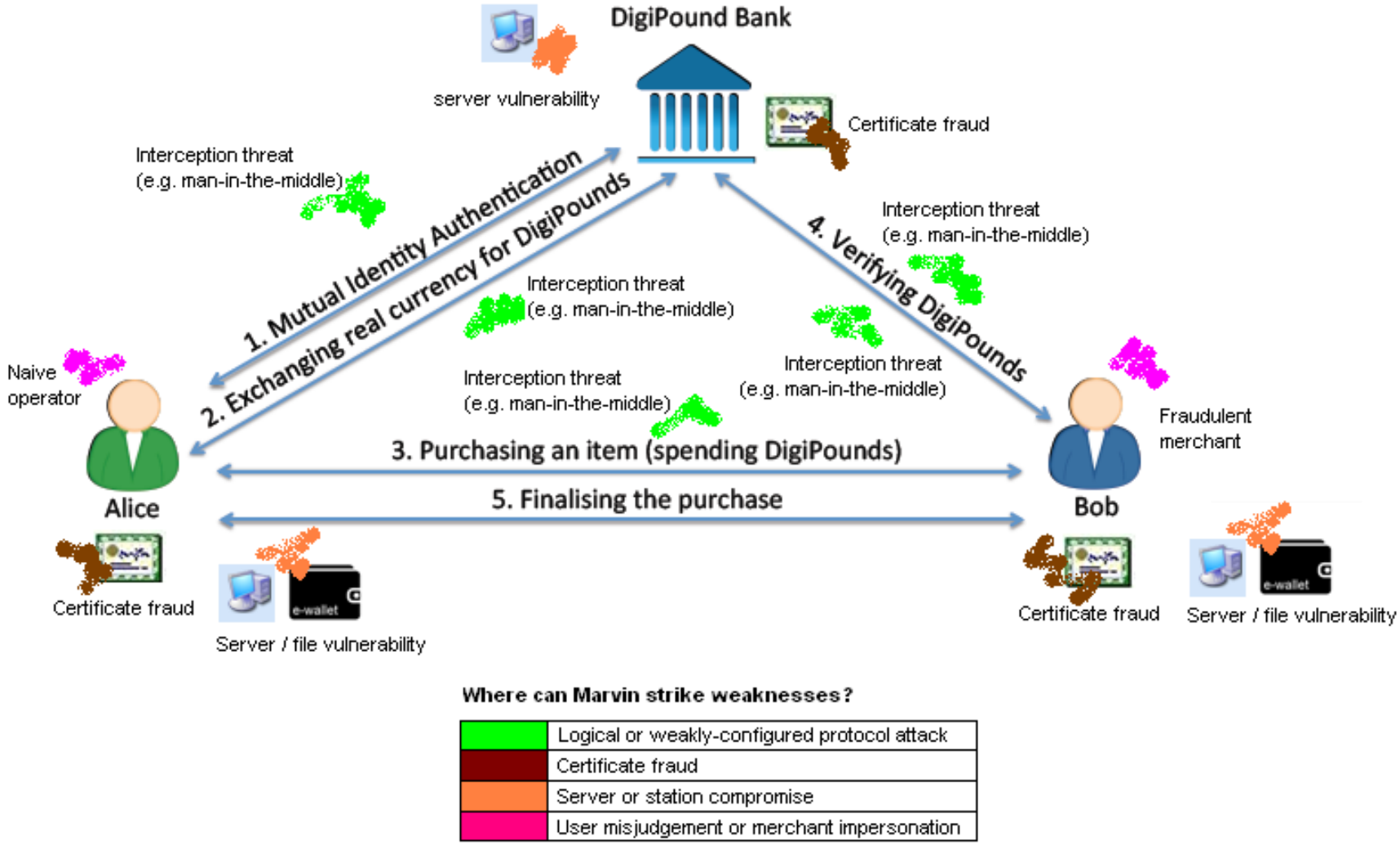


Figure 2: Points of risks, vulnerabilities and threats

(a.k.a. Marvin's attack strategy)

In *figure 2* we have used our threats and vulnerabilities from our previous register (which was based on a cause-effect diagram, where each threat or vulnerability stemmed from a requirement). Figure 2 compartmentalizes these risks to exhibit Marvin's points of attack in our scenario.

We incorporate the requirements analysis into the design of our protocols, and explore Marvin's points of attack more deeply with examples in *answer 6*.

2. Blueprints for a digital cash system in a real world context

For the design, we have been suggested several reasonable requirements for designing our DigiPound. Some of these may be elements of the DigiPound itself (such as the ID per unit), and some may be interpreted as necessities of the system surrounding it (such as protocol encryption per batch).

Settling on a system derived from existing implementations

Throughout my research of this answer, I have come across many different solutions. 'The electronic payment system' is a problem that has been tackled today via entirely online systems without Digital cash as an independent digital entity (but rather a virtual idea - PayPal 'holds cash' in an online account, but it is not a destroyable entity - it is merely a statement reading on a webpage!). Other ideas, such as DigiCash and Bitcoin, represent the money as an atomic bitstream - one that can be physically transferred and destroyed without bank intervention (although the bank may be required to legitimize the transaction). I was intrigued to conduct this initial research because I found common solutions no matter how different each system was. It gave me a good understanding of the question's abstract; correlation with realworld approaches, and ideas that can be used from each case. After I felt confident that I understood the problem and could visualize & describe it, I could then explain our system, then design the elements and the protocols. It was after understanding the system described and realworld implementations that I began to conduct my reading of the slides, books and papers.

I was tempted to write a section about these realworld examples (I could waffle for 20 pages easily), but none of the questions really required such a section for their answer.

So instead I made a diagram of my vision of the system which I drew from my realworld research, interspersing with reference and footnotes to similar or alternative approaches by existing e-payment systems throughout the report.

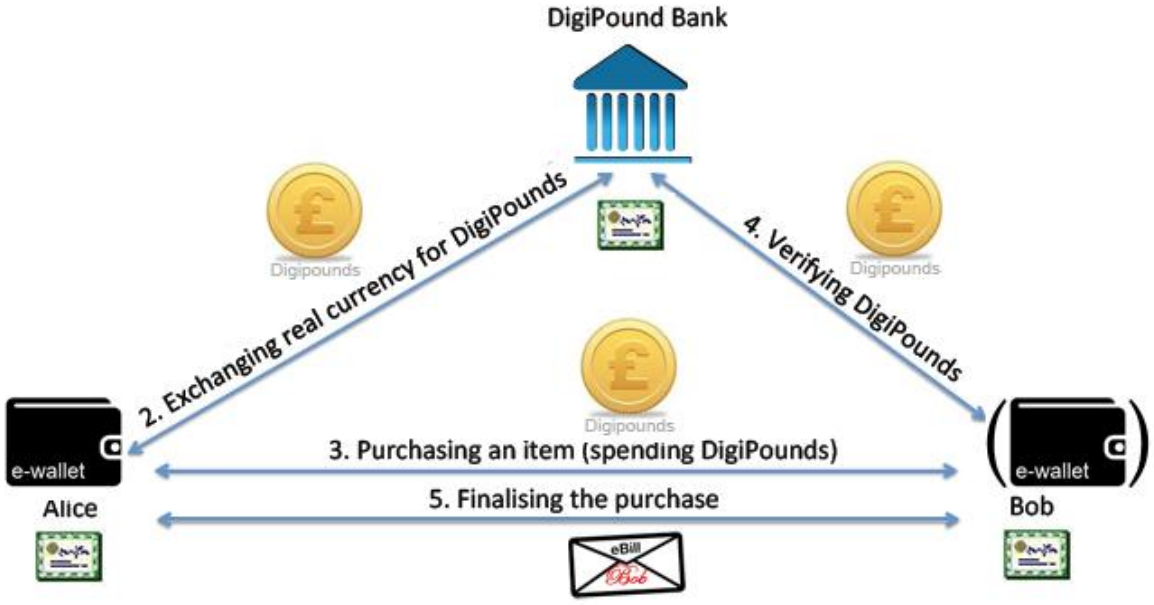


Figure 3: Transfer and storage of DigiPounds

The diagram above lays out our system, which was drawn from imagining the description.

Our system:

We have a digital currency system, the units of which are called the DigiPound.

DigiPounds are transferred to the recipient via protocols, and stored on their computer (in a securely-encrypted file known as an e-wallet).

To aid our description, transactions of multiple DigiPounds are bundled into an order. We do not wish to diverge into discussion of 'coin denominations' and 'change return' in our answer.

Coins cannot be created by the user (they are issued by the DigiPound Bank), but they can be destroyed or stolen (locally or in transit).

During the design phase, I had a look at some existing payment systems. No one payment system matched out scenario, although some came close. Our plan integrates the most appropriate from various ideas to suit the text.

Amongst these I looked at Mondex, NetBill, MintChip and others for approaches. I didn't wish to go off track, so you can see my two best inspirations in *Appendix A*.

We will make reference to all of these elementary components in the upcoming protocol descriptions, so it's good to get an understanding of them in the index over the next pages.

Justifying the design components

Where?	What?	Why?	How?
DigiPound itself	serial ID number	Unique identity number to identify each coin Assists R1 Prevents or identifies R3	Allows the DigiPound Bank to invalidate any DigiPounds that it has not generated. However, an attacker may still try to generate valid serial IDs (see *Digital signature* below). Allows the bank to identify Digicoins that have already been spent, using its record log
Money order component, Protocol cipher sute component	Hash function	To create a one way hash using a fixed algorithm (e.g. SHA-2) To provide integrity To make the message unpredictable and non-reversible To prevent the storage or transmission of plaintext passwords (effective when combined with a salt - see *answer 3*) Hashes can optimize performance	No two input messages will generate the same output, so if the hash is different when received, an error has occurred A good hash function will reduce collisions and make the output appear totally random in contrast with the input (which may have had many repeat phrases). This is more secure than a checksum, since the attacker can probably manipulate the checksum value ([3] Smith, 180) A good hash function can reduce the size of the input message, meaning faster subsequent encryption and transmission. They can also pad messages to become fixed length - useful for generating block keys (see *answer 3*) A hashed MAC is based on two symmetric keys (available to us)
Money order component, Protocol cipher suite component	MAC (a.k.a. MIC)	Creates a message authenticity code (used for verifying integrity, along with hashes) to verify the message has not changed. It requires the sender's key (Smith, 172) It is used for entity authorization in SSL (K Martin - see *answer 4*)	MACs are generated with a secret key algorithm. A hash function can only provide a strong degree of data integrity if combined with another mechanism to prevent the hash function from being manipulated ([4] Martin, p.205), so we introduce a key to the hash to create a MAC If the MAC can only be decrypted with a shared secret, then we know only a trusted party should have generated it. However, they don't provide non-repudiation (Martin, p.214) - either the sender or receiver could have generated it. Instead with PKI we use a signature (below) to verify the sender. A MAC can be created from a one-way hash function - encrypt it with a symmetric algorithm ([2] Schneier, p.456), like 3DES or AES See *answer 6: Quick theory* for how MACs prevent a number of attacks
PKI (i.e. DigiPound scenario infrastructure component) - servers and client stations	X.509 certificates	Provides a certificate signed by a trusted third party that is integral to creating signatures (below), allowing Alice to provide her public key, and to prove to the recipient she is sending the messages. Used in both SSL and CSL protocols (see *answer 4*). Revocable if the client checks a CRL list	User generates a public key with their private key (held in silicon, and never revealed), and sends it to the DigiPound Bank (Certificate Authority) as a Certificate Signing Request (CSR). The CA checks the owner's credentials, and signs it with their own signature (held totally secure - a breach would invalidate all signed certificates), turning the CSR into a valid issued certificate. The public key, provided via an authorized certificate (to Bob, or the bank, or to whomever Alice is interacting with) is a component for symmetric-key encryption of protocols (see *answers 3-5*)

Where?	What?	Why?	How?
DigiPound itself (bank signature) *Prototocol messages*	Digital signature	Used for showing the coin originated from the DigiPound Bank Used for verifying the sender is who they claim to be in protocol messages	If Alice encrypts a message with her private key, and Bob decrypts the message with her public key (found on her trusted certificate), then we believe that only Alice could have generated the encrypted message. Similarly, if a coin is signed by the DigiPound Bank's private key, the DigiPound Bank's public certificate will prove that it is not a fake. This is not a form of serial ID per coin (shown above) - it is just a blanket seal to indicate that it is authentic (see *answer 5*).
Protocol messages *e-wallet*	Digital encryption (by symmetric-key algorithm)	Used for encrypting a message such that only the intended recipient can view it	In contrast to digital signatures, if Bob encrypts a message to Alice with her public key, then only the intended recipient Alice will be able to decrypt it by her private key
Protocol messages	Nonce	Used for challenge-response authentication (see *answer 3*)	We dismiss timestamps (used in protocols such as Kerberos) because they create strict time-synchronization demands on included parties, which we cannot guarantee for Alice and Bob We prefer nonces - random (unpredictable) challenge numbers that are used once (preventing replay attacks) Nonces require two message exchanges to verify freshness, where a set time-window is used to consider if the returned nonce is still fresh (Martin, p.264). Timestamps would only require one message Nonce based authentication does not require clock synchronization, but it does require a capable pseudorandom number generator (so that it cannot be predicted by an attacker), which may cost in CPU resources
Protocol messages	Session key	Creates a short-lived symmetric key to encrypt the session	SSL uses a random session key. The session key must not be predictable, otherwise the SSL session will be compromised (e.g. SSL PRNG compromised by Wagner et Schneier, Berkeley (1997), http://www.schneier.com/paper-ssl-revised.pdf) They have high exposure and short lifetimes, i.e. a single session (Martin, p.341) We expect the authentication protocol to help derive a session key ([1] Kaufman, p.233) While the authentication step is expensive, we expect the session key to provide integrity protection and encryption for the rest of the session, dismissing the long-term secrets used in the authentication step (i.e. public-key) PKI is used to secure the session keys - the session keys are then used to secure the traffic (Schneier, p.33)

~~*Deducing security requirements by looking at system threats and vulnerabilities*~~

~~*Blueprints for a digital cash system in a real world context*~~

Protocol design:

3. A chosen authentication protocol between a client and bank (step 1)

"Designing protocols is an art in itself deserving its own handbook. In general we recommend you not design authentication protocols but instead adopt a standard, well vetted one from the many cookbooks."

Smith et Marchesini, Chapter 9.5.5, p.235

I have picked up a copy of *Applied Cryptography: Protocols, Algorithms and Source Code in C* (B. Schneier, 1995) as my cookbook.

Network Security: Private Communication in a Public World (C. Kaufman, 2002) was a valuable guide and complement.

Protocol design theory: Augmented encrypted key-exchange

The Encrypted Key Exchange protocol (Bellovin, Merrit) provides authentication between a client and server. They both share a password.

This matches our scenario, where Alice is the client and the DigiPound Bank is the server. We expect Alice to access her online DigiPound Bank account via a website portal, entering either a fixed password or key from an external keycard device (see additional optimization suggestions *in answer 5*), as a supplement to her certificate which we will explain in a moment.

Our first goal is to establish a secret session key with a client-server handshake. When both parties establish a secret session key, the handshake is complete, and we can start transmitting content securely.

In the most basic EKE protocol, Alice (A) and the DigiPound Bank (B) share password (P) which allows them to authenticate, and generate common session key (K) (Schneier, p.518).

I have re-listed this original handshake in my own words below:

(1) Alice holds a public-key/private key pair (her private key in silicon or bank issued keycard, and the public key in her DigiPound Bank issued certificate, generated by the DigiPound Bank)

The public key is encrypted (K') with a symmetric encryption algorithm, where P is the key. Alice sends the bank:

$A, E_P(K')$

(2) The DigiPound Bank decrypts this with shared P to obtain the message K'. It generates a random session key, K, and encrypts it with the received public key, using P as the key. The bank returns:

$E_P(E_{K'}(K))$

(3) Alice decrypts this to obtain K. Alice generates a random string, R_A, encrypts with K, and sends the bank:

$E_K(R_A)$

(4) The bank decrypts this to obtain R_A. It generates another random string R_B encrypts both with K and so sends Alice:

$E_K(R_A, R_B)$

(5) Alice decrypts this to obtain R_A and R_B. Provided R_A is the same one she sent in (3), she proceeds to encrypt R_B with K, sending the bank:

$E_K(R_B)$

(6) When the DigiPound Bank decrypts the message to obtain R_B, they can verify it matches the value they first sent. This means both parties are securely communicating with session key K.

(See Schneier, p.518 for a vanilla explanation)

So we see an authentication protocol with logical design. A particular benefit of this is that attacker Marvin is prevented from attempting to guess password P, because he is unable to test his guesses without cracking the public-key algorithm as well (Schneier, p.519). So presuming he cannot crack the public-key algorithm, he will have to revert to man-in-the-middle attacks instead (see *answer 6*).

So now let's choose a public-key algorithm for key exchange. The EKE protocol can be implemented with numerous public-key algorithms. To lead on to the next answer, we choose an implementation of EKE with Diffie-Hellman.

We note that a problem with the EKE protocol is that it requires both parties to store P. In the realworld most password based authentication systems should store a one way hash of the password (Schneier, p.52: Authentication Using One-Way Functions). R. Needham & M. Guy illustrated a principle that is put in practice by secure providers – that the host does not need to store the password, just deduce whether it is correct. When Alice sends her password, the DigiPound Bank compares a one-way hash with its previously stored value. The gives Alice confidence that if the DigiPound Bank is compromised she will not necessarily have to change her password wherever else she is using it. We also hope that the host is using a concatenated salt to generate the one-way hash, making dictionary attacks more difficult.

So to allow for this for this real world scenario, we use the Augmented EKE protocol which uses a one-way hash of Alice's password as the *superencryption key* in the Diffie-Hellman implementation of EKE: DH-EKE (Schneier, pp.519-521).

To re-implement our basic EKE protocol with Diffie-Hellman, we actually make it simpler.

K is now generated automatically, and two values g & n are set for both Alice (A) and the DigiPound Bank (B).

A series of modulo exercises now allows Alice and the bank to verify their random nonce challenges R_A (generated by Alice) & R_B (generated by the Bank).

(1) Now in step (1), Alice no longer encrypts her message with P, but rather sends the bank:

$E_{P'}(g^{rA} \bmod n)$

Encrypted with P', which is a hash of her password, and a chosen random nonce.

(2) The bank, knowing only P', which it cannot use to derive P, chooses R_B and sends:

$E_{P'}(g^{rA} \bmod n)$

(3) Alice and the bank now calculate the shared session $K = g^{rA} \cdot g^{rB} \bmod n$. Alice then proves she knows P itself (not just P') by sending:

$E_K(S_P(K))$

Schneier, p.521

Which is Alice's signature; i.e. she has performed an encryption with the value P that only she knows. When authenticating herself using her certificate alone (e.g. for CSL), she would use her private key to have performed the encryption as her signature, which the bank would then decrypt with her public key (which we spoke about in *answer 2, Justifying the Design components*). If authenticated with just the certificate alone (CSL), Alice would be at risk if someone stole her terminal (i.e. her phone), so in realworld deployments we use both a password and a certificate.

Alice and the Bank must agree on some digital signature scheme (Schneier, p.521) to generate the public key – which the case study provides us. The DigiPound Bank may have issued Alice an RSA or DSA Certificate – we expect RSA since it is based on Diffie-Hellman (not ElGamal), and reportedly faster[1] and more widespread[2].

If an intruder (Trudy or Marvin) retrieves the hashed DigiPound Bank password file from the DigiPound Bank servers, they cannot sign the session key unless they guess P. Therefore, the man-in-the-middle cannot imitate Alice.

[1] http://www.linuxquestions.org/questions/linux-security-4/which-is-better-rsa-or-dsa-public-key-12593
[2] http://security.stackexchange.com/questions/8343/what-key-exchange-mechanism-should-be-used-in-tls

Popular deployment libraries: SSL/TLS (handshake protocol)

We are going to consider the two aspects of the SSL protocol wrapper in accordance with the question layout - the handshake protocol to achieve mutual authentication, and the record protocol to transmit our DigiPounds by achieving a long term secure channel.

In this first authentication question therefore, we describe the SSL handshake protocol which we have chosen as a realistic choice for the given internet scenario. We also just chose a suitable key exchange algorithm (DH_RSA[2, 3] if the DigiPound Bank issues RSA certificates) to match Alice's certificate with the trusted DigiPound Bank.

In the second question, we move onto the record protocol component of SSL, and again choose a suitable cipher suite algorithm to suit consumer to merchant transmission.

*Please read my **note** at the top of page 23 to appreciate my terminology choice of 'SSL'!*

[3] List of cipher suites for SSL 3.0: http://tools.ietf.org/html/rfc6101.
List of cipher suites for TLS 1.2: http://tools.ietf.org/html/rfc5246

The SSL handshake

A top level view of an SSL handshake is simple.

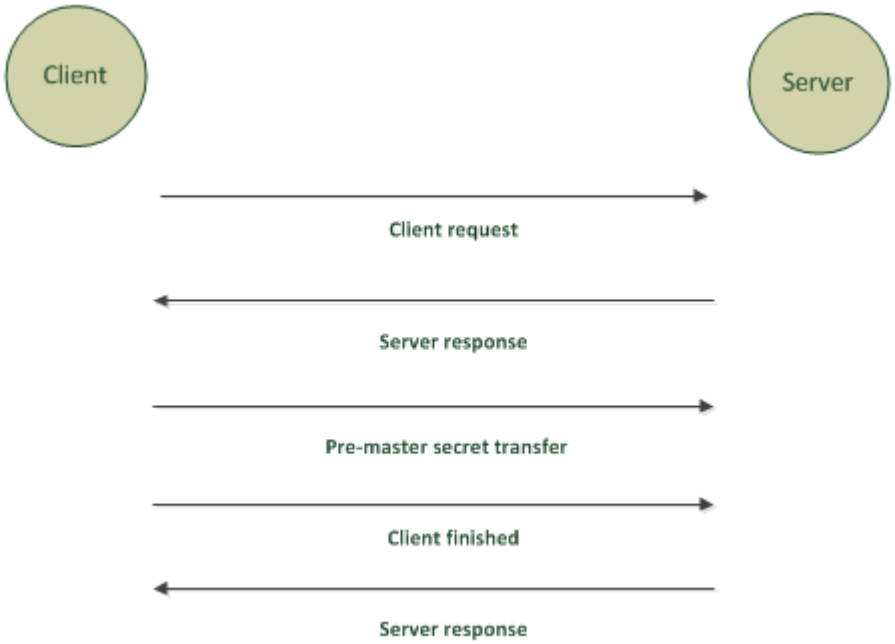


Figure 3: SSL simple handshake

Redrawn from Martin, 12.1.4 SSLprotocols, p.414

Since we are asked to provide authentication between Alice and her bank account, we can assume that this is a pre-verified, trusted relationship (unlike Alice and Bob). In fact, the bank has issued a certificate directly to Alice. Consequently we can make use SSL with Client Authentication, where the client is able to provide their issued certificate (message 4 below).

Alice provides a client request to the DigiPound Bank. She is initiating communication, and requesting permission for an SSL protected channel.

(1) Alice generates a session ID to act as unique identifier. This should not necessarily be random (see Kaufman, chapter 26: *Folklore*). She also generates a pseudo random number R_c, which is the nonce to guarantee freshness. She issues a list of cipher suites that her client station supports to the server.
(2) The server initializes by returning the sessionID, a server generated nonce R_S, a selected cipher suite from Alice's list, and a copy of the DigiPound Bank's public key certificate.

Alice is required to check the certificate's validity, and lookup the validity of any related public-key certificates if it is in a chain. It is the job of her browser to check any Certificate Revocation Lists to be sure[45].

(3) Alice now transfers the pre-master secret.

She generates another pseudo random number, K_p (the pre-master secret), encrypts it with the DigiPound Bank's public key and sends it to the DigiPound Bank server. This is unlike nonces R_c and R_s which are not encrypted because they merely provide freshness, and it would be an unnecessary expense to CPU resources. K_p however, is used to derive keys that provide a secure session.

Both the client and server then use a key derivation function to compute master secret K_M using K_p as key. They both then derive MAC encryption keys to use in the record protocol transmission. From this point, all exchanged messages are cryptographically protected (Martin, p. 415).

(4) Alice sends a copy of her DigiPound Bank issued public-key certificate to the server as a verification key for identification.
(5) Client Alice computes a MAC (e.g. using a hashed MAC (HMAC) with a hash function like SHA-2 and the secret key k_p) from all the transmitted messages so far and sends it to the server.
(6) The server checks the MAC, and the client's certificate. It then computes a MAC of all the transmitted messages so far, encrypts it and sends it to Alice.

To refer back to Martin, we have confirmed entity authentication:

1. The entity who sent the *Server Finished* message (6) must know the master secret K_M, since the final check was correct and relied on knowledge of K_M.
2. Any entity other than Alice who knows K_M must also know the pre-master secret K_P, since K_M is derived from K_p.
3. Any entity other than Alice who knows K_p must know the private decryption key corresponding to the public key sent in the *Server Response* message (2), since this public key was used to encrypt K_p, in the *Pre-master Secret Transfer* message (3).
4. The only entity with the ability to use the private decryption key is the DigiPound Bank server (since Alice verified the received certificate was legitimate). Alice's sent certificate (4) is also checked by the DigiPound Bank for validity.

[4] "Revocation in X.509 is at its core a list of certificate serial numbers that should no longer be trusted…Domain Validation: No major browser checks CRL or OCSP on these certificate types by default" *Defective By Design? Certificate Revocation Behavior In Modern Browsers, http://blog.spiderlabs.com/2011/04/certificate-revocation-behavior-in-modern-browsers.html*

[5] "Google to strip Chrome of SSL revocation checking", http://arstechnica.com/business/2012/02/google-strips-chrome-of-ssl-revocation-checking

5. The server is currently 'alive' because K_M is derived from fresh pseudo random values (K_p and R_c), generated by the client (preventing replay attacks).

Adjusted from Martin, SSL with client authentication, p.416

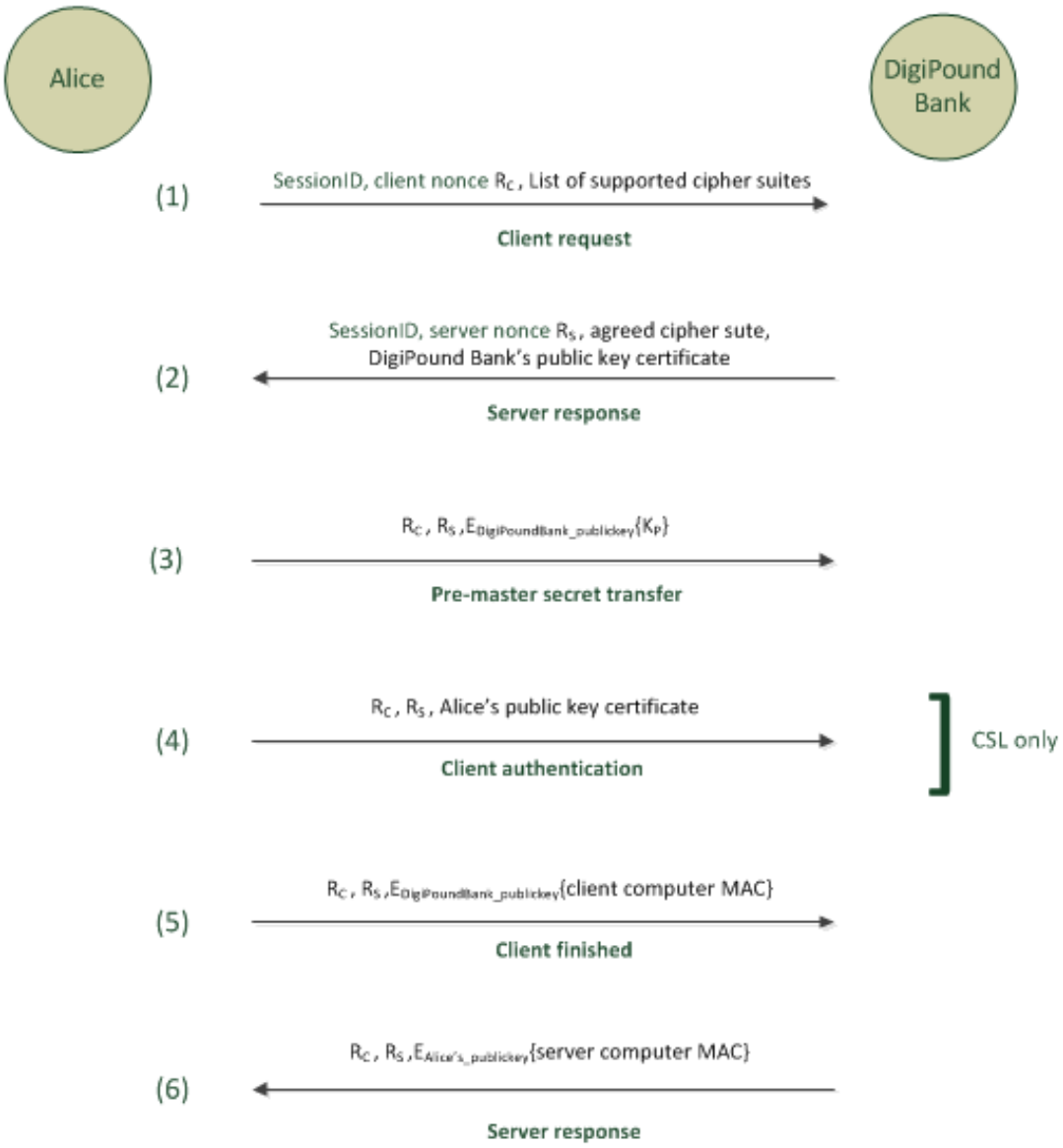


Figure 4: SSL handshake (with CSL)

Derived from Martin's figure and textual descriptions

Protocol design:

4. A DigiPound package transmission protocol between a bank and client, and a client to merchant (steps 2 & 3)

Assuming that the DigiPound Bank is satisfied that Alice and Bob are who they say they are (i.e. that the password they issued to set up their bank account is valid – for what e-bank communication would not require an initial password? And also that they have their original certificate), we come to a predicament where Alice wants to send Bob DigiPounds. The protocol can also send DigiPounds to Alice.

In the case of the bank transferring DigiPounds to Alice, the protocol could have made use of the stored password hash at the DigiPound Bank as well as the certificate, and perhaps a third method, software-based attestation (Smith, p.275) to allow 'multi-factor authentication' between Alice and the bank for DigiPound transmission. However, with the merchant we assume a more anonymous, less 'trusted' relationship.

Popular deployment libraries: SSL/TLS (record protocol)

Once authentication has been achieved with an SSL handshake, we require a secure channel to transmit our DigiPounds. Here we use the SSL record protocol (Martin, p.417).

Let's consider how this will work in our particular case.

We have identified that in order to transmit a transaction of DigiPounds, we cannot state that we wish to transmit *n* multiples DigiPounds of denominator *d*. Aside from making a very short message that could destroy the entire DigiPound economy if an attacker altered *n* (as well as allowing us to generate DigiPounds), we have established from question 2 that each DigiPound is a unique entity with a unique identifier.

Therefore we are transmitting a stream of DigiPounds, the message length depending on the value of DigiPounds and denominations used. It could be a large message.

We need to confirm the sustained integrity of the message after the handshake. SSL offers us the record protocol. The tactic for accomplishing this is splitting the message into encrypted blocks called *key blocks*. The handshake established the cryptographic data used to secure the session, including the symmetric session key, symmetric MAC keys and Initialization Vectors (pseudo random numbers used once – i.e. nonces). To generate the key block, a key derivation function uses the master secret K_M described in *answer 3*, along with the client and server nonces (R_c & R_s) also drawn from the handshake, and cuts it into blocks. By continuously verifying these blocks a tamper-free channel can be established for each session.

The SSL 3.0 record protocol specification can be explained in full detail (http://tools.ietf.org/html/rfc6101), but to give an overview:

Four symmetric keys are extracted from the key block:

K_{ECS} for symmetric encryption from the client to the server

K_{ESC} for symmetric encryption from the server to the client

K_{MCS} for MACs from the client to the server

K_{MSC} for MACs from the server to the client

The process for exchanging data from the Alice to Bob (client to server) is:

1. Compute a MAC on the data and other pseudo random inputs using key K_{MCS}
2. Append the MAC to the data and then, if necessary, pad to a multiple of the block length
3. Encrypt the resulting message using key K_{ECS}.

K. Martin, p.418

Bob decrypts using K_{ECS} and verifies the recovered MAC with K_{MCS}. SSL embraces key separation, where separate encryption and MACs come from the same master secret (the most important element of the handshake) but a different key is used for each transmission direction. This protects against reflection attacks (mentioned briefly in *answer 6*), and the cost is reduced since they are all derived from the common master secret. Sometimes for convenience SSL uses the master secret key as a MAC key during the handshake (Martin, p.420), but they should be distinct when valuable data is being transmitted, such as in this case. A great aspect of the SSL protocol is that it is application protocol independent; that means any application protocol or content can sit within it[6], and benefit from our chosen cipher suite.

Therefore, we extend our choice of SSL cipher suite for the record protocol. As RFC6101 tells us (almost identical text is used in RFC5246 for TLS):

"The SSL protocol provides connection security that has three basic properties:

The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES [DES], 3DES [3DES], RC4 [SCH])."

TLS 1.2 offers us different symmetric-key algorithms: (e.g., AES [AES], RC4 [SCH], etc.)

"The connection is reliable. Message transport includes a message integrity check using a keyed Message Authentication Code (MAC) [RFC2104]. Secure hash functions (e.g., SHA, MD5) are used for MAC computations."

(two properties listed, as TLS 1.2 dropped the third).

So that means we now need to choose a symmetric key algorithm & hash function for our cipher suite.

Starting from: DH_RSA

We might select:

(CipherSuite) SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

Why have we chosen SHA(-2) instead of MD5?

Because Carnegie Mellon University's Software Engineering Institute has stated that MD5 is insecure[7].

[6] "The SSL record protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL handshake protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL protocol transparently." SSL 3.0 (RFC 6101), http://tools.ietf.org

[7] "(MD5) should be considered cryptographically broken and unsuitable for further use" (CERT Vulnerability Note VU#836068)

Furthermore, we choose the SHA-2 (particularly, SHA-512 out of the SHA-2 set's choice of 224/256 or 384/512) as it is the current trusted hash algorithm utilized by the existing digital currency Bitcoin (with TLS, SSL, SSH) for its performance and security (replacing SHA-1 that has been retired[8]).

We could use 3DES which applies a DES cipher 3 times to each key block (we appreciate that it does not necessarily improve the security 3 times, and is more expensive than DES – but suitable for the type of data we are transmitting). We know it's secure and trusted – even Microsoft System Center Configuration Manager 2012 uses 3DES for password protecting user content and system data[9].

But we have something more exciting in mind for the banking world, and we are going take a look at advanced cipher suite options after this recap.

In *answer 3* we extended SSL with client authentication thanks to the DigiPound Bank certificate that Alice possesses, and the trusted relationship they share. SSL is just as capable of establishing sessions 'between strangers' (Martin, p.413), and that is the situation that we assume when Alice communicates with merchant Bob. If however Alice were also acting in a 3-way proxy to between Bob and the DigiPound Bank server, then Alice could verify that Bob is definitely a trusted merchant. However, for this DigiPound transmission protocol we wanted Alice to send straight to Bob. It's appropriate because in similar spending scenarios, the merchant may not use the same bank or certificate provider as Alice – the merchant will be considered a stranger.

[8] Federal agencies should stop using SHA-1 for...applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010" NIST's Policy on Hash Functions, National Institute on Standards and Technology Computer Security Resource Center (2009) http://csrc.nist.gov/groups/ST/hash/policy.html

[9] Microsoft TechNet Technical Reference for Cryptographic Controls Used in Configuration Manager (January 2013), http://technet.microsoft.com/en-us/library/hh427327.aspx

Cipher suite options and optimization with ECC

A note:

The latest revisions, TLS 1.2 and SSL 3.0 are two different entities (to be more specific, SSL is the defacto standard for secure internet data transmission, and TLS is based on it; both are maintained by IETF). We have referred to them as SSL until now (I took a leaf out of our academic textbooks, specifically Martin (Oxford) page p.412: "We will choose to treat SSL and TLS as the same protocol and refer to this protocol as SSL."). However, we now come to a point where we want to choose some advanced cipher suite combinations that SSL 3.0 does not provide but TLS 1.2 does. Intended as an eventual successor[10], TLS 1.2 is backward compatible with SSL 3.0, but additionally offers us AES and ECC in RFC5246 (http://tools.ietf.org/html/rfc5246) which SSL's RFC6101 does not (http://tools.ietf.org/html/rfc6101).

Let's show two contrasting SSL cipher suites and take our pick:

Cipher suite	Encryption algorithm	Key length	Hash algorithm	Key exchange	Certificate
01	RC4	40 bits	MD5	RSA	RSA
02	AES	256 bits	SHA-2	EKE-Diffie-Hellman	RSA (or DSS)

Table inspired by: CICS Transaction Server for z/OS V3.2: Security: Security for TCP/IP clients: About security for TCP/IP clients: Support for security protocols: Cipher suites (for SSL connections) http://publib.boulder.ibm.com/infocenter/cicsts/v3r2/index.jsp?topic=%2Fcom.ibm.cics.ts.doc%2Fdfht5%2Ftopics%2Fdfht5nv.html

Cipher suite 01 is deprecated: MD5 is vulnerable. We justify our preference of AES-256 as being a strong choice for 2013 – it performs well on low resource devices such as 8-bit keycards (source: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Performance).

So what about the future? We've read a lot of good things about ECC, and especially about its performance and integration:

"because it gets more security "bang" for computation and memory expenditure, ECC is receiving more attention in embedded systems applications and resources-constrained devices, such as PDAs and cellphones. We urge curios readers to refer to Schneier … for details on these" Smith, p.180

"Many public-key algorithms such as Diffie-Hellman can be implemented with elliptic curves." Schneier, p.480

[10] SSL/TLS,Mozilla.org, http://www.mozilla.org/projects/security/pki/nss/ssl/

Kaufman follows his section 'How secure Are RSA and Diffie-Hellman' (Chapter 6.6, p.178 - this is really just a long verse of song!) with discussion of ECC. Like Schneier, he is a proponent, stating that it is believed to be secure with smaller key sizes to optimize performance (referring to K Martin's key length equivalence table (Martin, p.177) we see an RSA modulus of 1776 bits requires only 192 bits with ECC for equivalent security. For RSA, a bit length of 1024 is standard and 2048 is safe, yet for ECC 192 is standard and 224 is safe). Also, in some crypto schemes, modular based algorithms like Diffie-Hellman can be modified with ECC to create ECC Diffie-Hellman as a drop in replacement (instead of choosing the TLS specification's DH_RSA cipher suite we mentioned earlier, we pick ECDHE_RSA). Provided the DigiPound Bank can afford the patent licences, I would like to recommend they do that here. Realworld programmer forum implementation discussion suggests device and platform compatibility may hinder Alice (turn to Appendix B for the full discussion) – although if the bank uses ECC with RSA certificates, then Alice's latest smartphone should support it (and the bank should enforce she runs transactions on up-to-date software/hardware as a rule of good practice).

Therefore, for our final cipher suite choice we choose not:

SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

nor (in TLS 1.2, RFC5246):

```
TLS_DH_RSA_WITH_AES_256_CBC_SHA
```

but the Elliptic-curve Diffie-Hellman key exchange algorithm with AES encryption (RFC4492[11]):

```
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
```

..because AES gives us better performance than 3DES (a better choice for devices such as smartphones[12]). AES supersedes DES, the foundation of 3DES (NIST,2002[13]).

[11] Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS), IEFT (2006)

[12] *"DES was originally designed to run in specialized hardware and is considered "computationally expensive" on general-purpose processors. AES was designed to run efficiently on a variety of processors. AES should give you better performance (and more security to boot)--test reports I've seen on various Nokia platforms bear that out."* https://www.cpug.org/forums/ipsec-vpn-blade-virtual-private-networks/49-difference-between-3des-aes.html

[13] "NIST reports measurable success of Advanced Encryption Standard", *Journal of Research of the National Institute of Standards and Technology* (2002)

Protocol design:

5. A protocol for verification of the DigiPound package, between a merchant and bank (step 4)

In the first 2 of the 3 cryptology and protocol questions we've looked quite exhaustively at textbook explanations for tried, abundant technologies on the internet (namely SSL utilizing challenge-response authentication), documented by our favourite 3 authors!

We will use this third question to look at academic papers (particularly, those based on Chaum's work) and existing realworld implementations that have evolved from this to achieve dependable currency verification, and finally just confirming these details with the reliable literature of Schneier.

We already made reference to some realworld solutions in *answers 1 & 2* when we designed our system - we will tie in those implementations as well.

A proven currency verification solution to meet our requirements

We come to the problem of verification. We recognise that Alice is sending Bob DigiPounds directly, so now Bob needs to confirm their validity.

We wish to confirm that:

We are checking for doublespend with the DigiPound Bank;

We are checking the validity of the coins (i.e. they are not fraudulent) with the DigiPound Bank.

The benefit of transaction anonymity:

Alice may wish to cover her transaction fingerprint(s) - consider a legal privacy precedent where the merchant could view past transactions (i.e. how the client had generated the money), and therefore claim personal insight into Alice's dealings. Like physical coins, spending digital money should not leave the same auditable trail that the credit card industry allows;

Alice should only pass on the details that she wishes to share with Bob (i.e. not necessarily everything the bank knows about her) - for they could be misused in an identity profiling attack (or personalized spam);

There are also numerous problems with true anonymity if one cannot reveal the system's abusers. We discuss this briefly in *answer 6*.

Ideally the goods can be released as soon as the transfer is made, meaning that the cash can be verified immediately. This is easiest with a trusted third party (Smith, p.372).

Scenario recap:

We require the bank to verify the legitimacy of our DigiPounds without knowledge of who they originated from (i.e. Alice, to whom they originally dispensed these pounds) or of the transaction.

We know that each coin has a serial ID.

We know that the bank keeps "*a record of the spent DigiPounds*".

Solution 1:

We deemed the diagram suggests the source of all DigiPound generation is the DigiPound Bank (unlike some real systems like Bitcoin, where users can mine their own coins).

The simple answer would be to say that we know each DigiPound:

a) Has an ID

b) Is sent back to the DigiPound Bank in step 4, which can then lookup its database of

i) pre-generated DigiPounds to check it exists

ii) record of spent DigiPounds to check if Alice's DigiPounds have not already been used.

Assuming that DigiPounds are a 'one-use-only' entity, the DigiPound Bank record could hold a timestamp of when each serial ID was verified, and reject any subsequent attempts at expenditure for that serial ID, thus preventing double spend.[14]

However, this answer is flawed because Marvin could figure out the format or sequence of serial IDs, and begin to generate his own DigiPounds. It also means the bank must keep track of database i), which our description gives no indication of.

Better we incorporate a feature of *answer 2* – we want the bank to sign every DigiPound it generates. This gives us a performance benefit as well.

Solution 2:

We know that each DigiPound:

a) Has an ID

b) Is sent back to the DigiPound Bank in step 4, which can then lookup its database of

i) record of spent DigiPounds to check if Alice's DigiPounds have not already been used.

c) Has been signed by the bank.

We need to have the DigiPound Bank generate the DigiPound, and sign it to prove legitimacy (otherwise Marvin could generate his own unsigned money). The bank would sign each coin with their private key.

[14] This is how Bitcoin prevents doublespend. It has no central server record of spent serial ID against time; it instead hashes the coin's history, including expenditure times, to each coin. "*Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner*" Wikipedia

If the DigiPound design did not include a bank signature (Solution 1), it means Marvin could generate lots of DigiPounds and flood merchants, and consequently the DigiPound Bank server, with fake DigiPounds easily (more fake than legitimate). But if the coins are signed (Solution 2), Bob can immediately see that it has been signed by the DigiPound Bank – using step 4 to check for double spend alone. If Bob can reject fake coins immediately himself (confirming the bank signature by applying the bank's public key), he can offer Alice (or Marvin) quicker failure notification, and save the DigiPound Bank from being flooded with fake currency requests. All he needs to do is store a copy of the DigiPound Bank's public key certificate.

Now the solution above works, but it means that coins can only be used once. Surely if Bob is verifying the DigiPounds, he can keep them as the second owner?

Solution 3:

The easy answer would be to say that the DigiPound Bank stores each coin's current owner, along with the serial ID and expenditure timestamp. All transactions would have to be in realtime, and when Bob verifies Alice's attempt to spend, Bob is immediately assigned as the coin's current owner. However, in the realworld, as in this description, the bank should not be able to look up the coins current owner by keeping a record.

Perhaps instead, we could mimic Bitcoin:

1. The DigiPound Bank holds the serial ID of the coin, the time it was spent, and a hash created by the *public* key of the owner;

1: To transfer, Alice unlocks the coin with her private key, and then encrypts with Bob's public key. Consequently, the coin can only be 'unlocked' by Bob, who can then spend it.

2. Bob unlocks the coin, so he can view it. He views the unique Identifier (a non-sequential hash), and re-encrypts the coin with Alice's public key.

Bob then sends the Bank the coin's unique Identifier (proving he must be the assigned owner that Alice designated), the coin re-encrypted with Alice's public key, and the coin re-encrypted with his public key:

$$SSL\{serialID, E_{Alice's\_publickey}\{coin\}, E_{Bob's\_publickey}\{coin\}\}$$

3. If the serialID and first hash match a given entry, The bank updates the coin record, keeping the serial ID, replacing time of spend, and replacing Alice's public key hash with Bob's public key hash, for he is the new owner.

Coin's serial ID Non-sequential unique identifier generated by encrypting a sequential number with the DigiPound Bank's private signature	Time of ownership (spend)	Coin hashed with current coin-owner's public key
$E_{DigiPound\_Bank\_privatekey}\{01\}$	2013-2-1.15:01:89	0xcf83e1357eefb8bdf1542850d6

Now if Alice were to try to spend it with another merchant (by re-encrypting the coin with the second merchant's public key), the DigiPound bank spent record would state that Alice is no longer the owner (because Bob has already updated the record).

So it is basically the same system as Bitcoin, only Bob uses the third party to verify that the last owner assigned to the coin was definitely Alice, and that he is authorized to become the new owner. The bank cannot trace Alice, or the new owner Bob – it just stores a public-key encrypted hash for lookup (which only Bob can unlock), and a transaction time that the new hash owner (Bob) claimed ownership, to deduce that Alice didn't try to pass it to two different merchants at once.

Update – fixing a collusion attack:

The above solution appears to work, but what if Merchant 2 was in collusion with Alice, and consequently applied the new owner's public key to the spent coin for verification and claimed ownership?

Therefore, the bank must offer the 'merchant claiming ownership' a challenge to prove they hold the certificate with the public-key used to encrypt the coin.

Provided they can prove they hold:

1. The public key used to encrypt the coin to create the owner's hash;
2. The private key that corresponds to that public key (either by certificate lookup, or more likely by offering a second 'encrypt-rencrypt' challenge)

… then they may take ownership of the coin.

Additional discussion: the option of true anonymity with Chaum

While the solution above works to provide coin authenticity and prevent double spend, with a voluntary mechanism of anonymity and re-use that is currently very popular (only using a peer-to-peer-network as Bitcoin does, not a central server), I would like to discuss research based on Chaum's work[15]. It is more difficult to relate, because we have already established:

We know that each coin has a serial ID.

We know that the bank keeps "*a record of the spent DigiPounds*" (for which it requires a serial ID).

We know that all coins are originally bought from the DigiPound Bank (in all the blind signature examples I encountered, it was Alice who was trying to get the bank to sign money that she herself had generated, without allowing the bank to see the content).

I chose the 3 solutions above because I was more confident with the *Bitcoin method*, and I could utilize the 'spent record' our description had mentioned. Nonetheless, here is some 'just for fun discussion' to try to exhibit my other reading.

An alternate path

In solution 2, what is to stop the bank from looking up the serial ID when Bob verifies it (step 4), and tying it back to its record of original purchaser?

Something like this may simply come from data confidentiality laws – i.e. they *promise* not to keep a record of the coins they sold to Alice, or they *promise* not to look up this coin by tracing the serial ID to Alice. Provided the bank didn't keep any record of Alice's name against coin serial numbers when they sold them to her, they can't trace her coin back to her when Bob verifies it for a transaction.

Instead we tried to obscure the owner's identity in solution 3, each owner re-signing the coin with the next owner's public key to transfer ownership (after they had proven they could unlock it first).

Now, for no other reason than "*we can, and we have"*, let's consider Blind Signatures.

[15] Chaum D. (1982), Advances in Cryptology Proceedings of Crypto 82 (3): 199–203, http://www.hit.bme.hu/~buttyan/courses/BMEVIHIM219/2009/Chaum.BlindSigForPayment.1982.PDF

With a blind signature scheme signers sign data that they cannot see. The step at which this is most appropriate is Step 2: when they (the bank) sign the DigiPounds to Alice, we don't want them to have any record of who they gave those DigiPounds to. Again, it's a different scenario, because since the bank is the one generating the DigiPounds, the bank must know that they are legitimate – it would simply pick pounds at random from its repository and send them to Alice. When Alice bought DigiPounds, the DigiPound Bank signed the DigiPounds to verify their authenticity – that has already been established (differing from most analogies where we assume that Alice is generating her own money - or carbon cheques - and asking the bank to sign them using the cut-and-choose method and a blinding factor). The idea we are really aiming for (if we are to mimic most modern Blind Signature Digicoin uses) is to have the bank be able to reveal Alice if she doublespends, by using secret splitting.

Chaum's solution used secret splitting and bit commitment (committing a message, but not revealing it until later) to address doublespend (Smith, p.374). A second benefit is that signing potentially allows a batch offline solution (in fact, it was the inspiration for having the bank sign each DigiPound in solution 2 – we discuss this afterwards in the *5.5 Optimizing verification* section), so that Bob can verify the legitimacy of each DigiPound as soon as it is received, yet have the bank check for doublespend later. If in verification he finds it has been doublespent, the bank can reveal the abuser.

Kaufman gives clearest explanation:

"If it is desired that the group membership server (DigiPound Bank) should not know which key (DigiPound serial ID) is associated with which member (Alice), the group membership server (DigiPound Bank) could do a blind signature, in which …(the DigiPound Bank) signs something without knowing what he is signing! Assuming you (Alice) want to be able to use your privileges as a member of the group (Alice, Merchant Bob, DigiPound Bank) without anyone being able to know what individual you were, this feature would be useful. With blind signatures, ..(the DigiPound Bank) does not know which keys belong to which members (consumers or merchants – the DigiPound Bank may have kept a log when it first sold Alice her DigiPounds, but it mustn't know where she spends them), and so cannot divulge this information."

Schneier offers us 4 protocols that progressively improve upon an implementation of an anonymous banking scenario (6.4, Digital Cash, p.140). The first introduces the blind signature cut-and-choose analogy, the second addresses the doublespending problem, the third introduces a uniqueness string (our serial ID number) to the DigiPound Bank records but is vulnerable to fraud. His fourth protocol uses splitting to identify Alice only if she is committing fraud.

Now, many online papers offer us some very detailed algebraic explanations of Alice, Bob and the bank in this triangle. I don't feel confident in quoting their proofs because I cannot verify if they hold true, although Cattani et al.[16], p.9 certainly seems to hit the button (it appears to offer a model

[16] Cattani et al. (2004) Digital Cash, http://www.cs.bham.ac.uk/~mdr/teaching/modules03/security/students/SS4/DigitalCash.pdf

answer but I cannot use it as the foundation for mine). I will stick to modeling on Schneier's fourth protocol which I have more confidence in (and it correlates with all of my literary sources - and since the foreword is by Chaum, I'm guessing it's correct!). He tells us that:

(1) Alice can prepare the transaction order containing the DigiPounds for a given amount (generated by her e-wallet software) with a random uniqueness string distinct to each order;
(2) The money order contains some notion of Alice's identity for secret splitting – whatever Alice is willing to divulge if the bank accuses her of fraud (the DigiPound Bank may have its terms). She splits these into pairs using the secret splitting protocol, which create n pairs of identity bit strings.
(3) Each identity piece is committed using the bit-commitment protocol.
("We can address this double-spending problem by using secret splitting and bit-commitment. When preparing a dollar bill (DigiPound) for blinding, Alice splits her identity into two pieces and bit-commits to each piece. (So, her dollar bill, before blinding, includes both the bit-commitment as well as the serial number)",
Smith, 14.1.5 Digicash)

Each order generated by her e-wallet contains:
Amount (float)
Uniqueness string (long integer)
Identity string, I1 = (I1L, I1R) (pair)
Identity string, I2 = (I2L, I2R) (pair)

(4) Alice blinds her money order using a blind signature protocol (uses RSA):
Bob has public key *e*, private key *d*, and a public modulus *n*. Alice wants Bob to sign message *m* blindly
a. Alice chooses a random value *k* between one and *n*. She blinds *m* by finding:
$t = mk^e \bmod n$
b. Bob signs t:
$t^d = (mk^e)^d \bmod n$
c. Alice unblinds t^d by finding:
$s = t^d / k \bmod n$
d. $s = m^d \bmod n$

(Schneier, p.550)

(5) She sends this order to Bob, who sends it to the bank for verification (note we accept the DigiPounds were *already signed by the bank* in the exchange for steps 1 & 2. Most descriptions we see of this trio involves Alice generating cash and asking the bank to sign it, but we can skip this step here because it was the bank that sent Alice the money initially – *we just want Bob to be able to verify it*). However, we can elaborate step 2, in saying that the bank handed Alice a *blind* money order. That is, they gave Alice a number of DigiPounds for the amount

she purchased, but they *did not* log each serial number they gave her (the order was blinded by Alice's certificate). As Schneier says, "*Alice unblinds the money order and spends it with the merchant*".

(6) The merchant verifies the bank's signature.
(7) The merchant asks Alice to reveal either the left or right half of each identity string on the order, which she must do.
(8) The merchant sends the money order to the bank (step 4).
(9) The bank verifies Bob's X.509 signature.
(10) The bank checks its database to check the uniqueness IDs (DigiPound serial IDs) have not been deposited before. If not, Bob's account is credited and he can continue with Step 5. The bank records the used serial IDs and the half of identity information.
(11) If the serial ID for the DigiPound is in the database already, the bank declines the coin. It also then compares the identity string on the coin with the one in its database. If they are the same, the bank deduces that the merchant copied the coin. Otherwise the bank deduces that it was the person who bought the money. The bank compares the opened halves of the identity strings, and if Alice tried to spend twice, they XOR the two halves together (one from the coin and one found from the bank database) to reveal her identity.

    This detects Alice's attempt to cheat by revealing her identity from the coin information;

    It means the merchant can't deposit the same coin twice as the bank will notice, and he can't blame Alice, since only she can open any of the identity strings.

    Neither can the bank make the connection of whether the coin it accepted from the merchant was from Alice even if it keeps complete records of every transaction, since the blind signature protocol covers her from steps (4)-(5). But it can reveal her if she double spends.

    There is also no way for the bank and the merchant to collude and reveal Alice. The only way is by XORing two identity strings (Cattani, p.12: illustrated in Appendix C).

(Schneier, p.144)

The only chance for abuse here is Marvin. If he snoops the communication between Alice and Bob, he can cash the digital money at the bank (or e-cash exchange) before merchant Bob. When Bob tries to cash it later, he will be called the cheater. Alternatively, if Marvin copied the DigiPounds (file) from Alice's computer before she spent them, the bank will claim Alice is the culprit. The e-wallet requires the same protection as a real cash wallet, which may not be practical in the smartphone world (as discussed in *answer 6*).

5.5 Optimizing verification

Let's take a quick look at optimization in this case:

We have the option of performing offline transactions which we mentioned earlier. In this online model we are checking as quickly as possible so that Bob can finalize the purchase – this requires bank verification. Bob will need to expend bandwidth on every customer transaction between him and the bank, as well as CPU costs for encryption per message.

The bank might issue Alice a trusted keycard device[17] or client computer program[18] for software based attestation. Basically Bob trusts that Alice's DigiPounds are authentic (or that Alice's client terminal has a verifiable, trusted relationship with the bank), so he semi-processes the order without having to check with the bank in realtime. He then sends the bank the batch message of all the orders he received from Alice that day (and perhaps all the other customers that placed orders, into one merged message), and sends them to the bank each night during a quiet period. While it cannot detect doublespend immediately, the bank could insure Bob against all abusers, and identify and chase them via the secret-splitting lookup. Or it could be that Bob checks for doublespend with the bank each day at 5pm, and ships at 6pm, notifying the customer of a payment error later in the day. Even if Bob verified the keycard/software transaction generated 'key' with the bank in realtime, we are still finding his CPU burden reduced by Alice's hardware.

We would also consider that an alternative model to our triangle, where Alice verifies directly with the Bank before having money sent to Bob's account, would lighten the amount of verification his server needs to do. This is similar to many merchant websites that utilize a Bank Authorization/Payment Gateway via a website iframe, and closer to how PayPal works in sending money to the trusted third party and then notifying the recipient. Her own browser/network connection is communicating with the bank through the iframe[19], and Marvin's website simply receives a short Boolean flag to indicate whether it was successful.

If the DigiPound Bank issued Alice and Bob remote attestation software, Bob's software might use it to verify Alice as a trusted party with less resource expenditure. Remote attestation offers clients different ways to prove they are who they say they are (Smith, p.444) – so with our different triangle model, Alice's remote attestation software works with her e-wallet to verify with the bank that her coins have not been doublespent before releasing them to Bob. If Bob's e-wallet sees that Alice is a remotely attested client as well, he trusts any coins that he receives through a secure channel between the two attestation clients have already been verified for doublespend (we knew already that they were legitimate coins because they were bank signed).

[17] https://www.paypal.com/us/cgi-bin/webscr?cmd=xpt/Marketing_CommandDriven/securitycenter/PayPalSecurityKey-outside

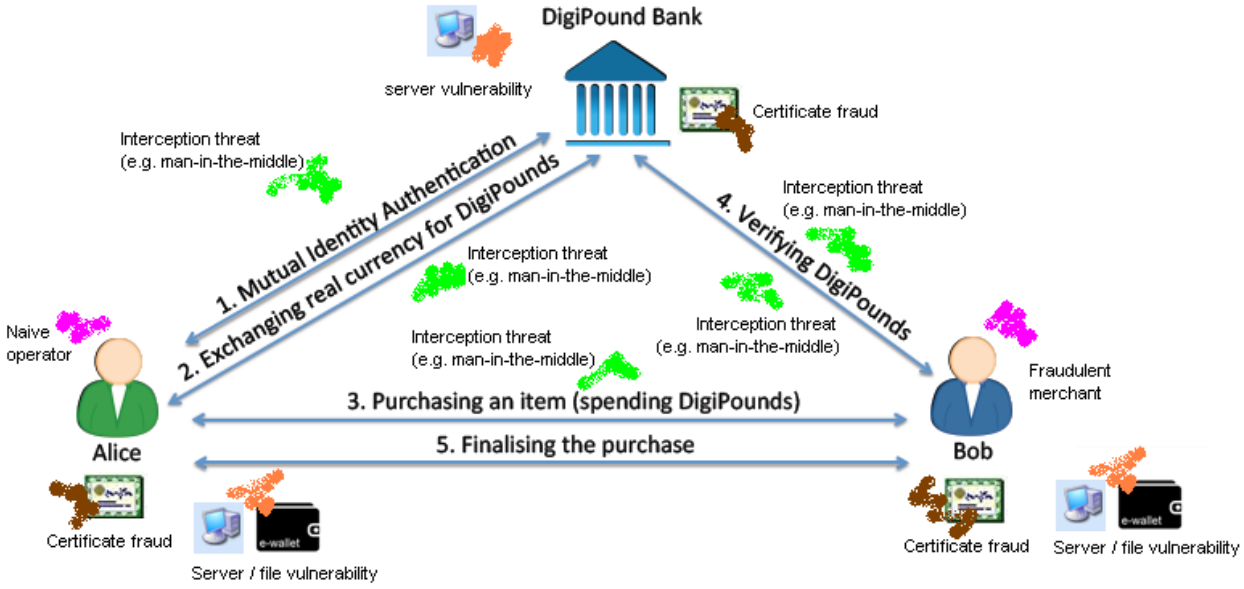
[18] http://www.natwest.com/personal/online-banking/g1/banking-safely-online/rapport.ashx

[19] Although how do we know that the iframe is not a fake generated by Marvin? Perhaps it could ask a choice of security questions that only Alice and the Bank could both know.

6. Modern attacks on real world infrastructure

Mobile WiFi infrastructure and carefree user habits introduce a number of vulnerabilities to internet based scenarios.

Let's look back to our *Points of risks, vulnerabilities and threats* diagram in *answer 1* (*figure 2*), adding some real examples.



There were 4 colours:

I		Terminal station exploit (malware at Alice's computer or smartphone, or Bob/DigiPound Bank's server)
II		Web application exploit (the server is well maintained, but the Bob/DigiPound Bank's software or website is vulnerable to exploit, e.g. SQL injection)
III		Protocol/network infrastructure exploit (vulnerable to Marvin the protocol cracker and Eve the evedropping snooper);
IV		Naive operator error (the Digbank has no fault, but Marvin tricks Alice by spoofing the DigiPound Bank server certificate, so Alice connects to the wrong server)
V		Naive operator error (the Digbank has no fault, but Marvin tricks Alice by spoofing a WiFi connection which he snoops, so Alice is being monitored)

Quick theory

I could go on about numerous protocol attacks that could afflict our network protocols, but as the question says, we have chosen evolved network protocols that are designed to have countered these ingenious circumventions. One flaw that may still remain however, is configuration - we might use a redundant hash algorithm like MD5, or choose a PK encryption algorithm that is subject to a particular attack, like RSA (again, we've intentionally avoided this, as an example will illustrate below). Rather than just recite the methodologies for attacks (which I will mention, but not quote ad nauseum, for they are available from Schneier chapter 11) it is more fun and valuable to come up with realworld infrastructure vulnerabilities that would afflict our scenario (as the question encourages us), and look at some actual attacks that have affected the web infrastructure which the DigiPound scenario particularly could suffer from.

Having said that, it is worthwhile to have a quick theoretical overview to illustrate an understanding of these attacks, before launching into some exciting application.

Just like SSL, there are two opportunities for interception. The first is during the handshake or authentication phase, where the user is tricked into establishing a session with Marvin (who is acting as the server). The second is Marvin's attempt to steal the session keys after authentication (known as session hijacking).

An interesting version of the man-in-the-middle is called the Chess Grandmaster (Smith,p.233):

Marvin impersonates Bob to Alice;

Marvin impersonates Alice to Bob;

When the real Alice tries to authenticate, Marvin establishes a successful session with Alice (even though he doesn't hold any of the data she expects), and uses Alice's data to establish Alice's intended session with Bob! He is then juggling two sessions (a reflection attack), while both Alice and Bob think they are legitimate.

This Chess Grandmaster attack would be quite useful in the Alice-to-DigiPound Bank communication (step 1 & 2). Marvin is stealing Alice's credentials (login and credit card), while using them to buy funds from the DigiPound Bank - Alice hands over yet receive nothing.

In the case of Alice-to-Bob (step 3), Marvin would be taking the DigiPounds that Alice sends to Bob. Again, Alice hands over yet receives nothing.

In this case Marvin doesn't expect to need a parallel session with Bob, since Bob will not provide anything until he has received and verified the DigiPounds.

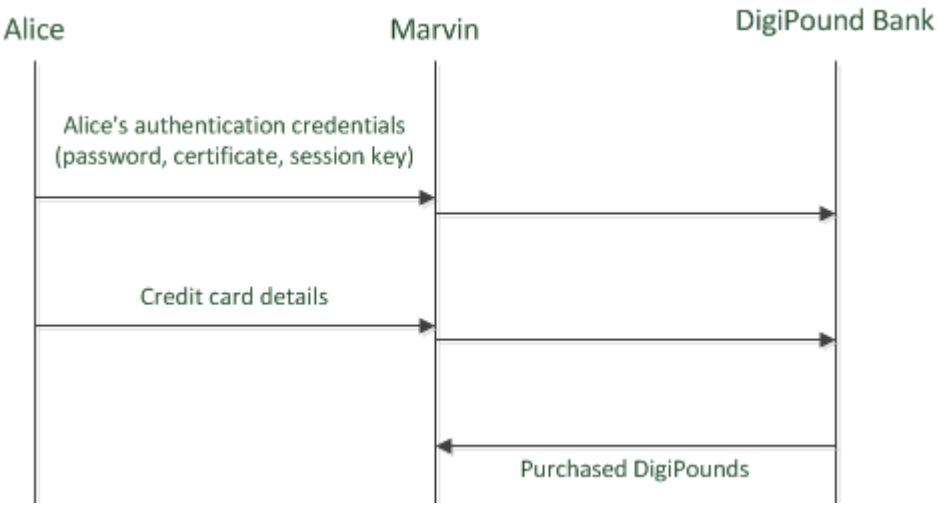


Figure 5: Chess Grandmaster attack

In the case of session hijacking, we take great care to ensure our session keys maintain integrity, so we can have two keys, one for a MAC and one as a key to the encryption algorithm. They are short-lived (derived from temporary data) to guard against replay attacks.

There is also such a thing as the oracle attack – consider if Marvin tricked one of our 3 actors into answering authentication queries by repeatedly starting and abandoning authentication sessions (Smith, p.235).

It's worth noting that in our design phase *(answer 2)* we spoke about the importance of the MAC, and I would like elaborate on the usefulness of that component here. It prevents a number of our theoretical attacks. For example:

A CBC-MAC prevents unauthorized insertion, modification, or deletion of part of a message;

Attacker Marvin cannot insert a false message (he can try to insert it, but without the symmetric key he cannot compute the MAC - however, if he compromises our session key (as that is often used to generate the MAC) then he can.

You cannot persuade the receiver that the message was sent from someone other than who it was, since a CBC-MAC (or any good MAC algorithm) provides origin authentication (Martin, p.212). Without the key for the hash, the attacker just has to keep guessing. If the length of the key is large, e.g. 256 bits for AES, then the likelihood of guessing it is unlikely (as described by the birthday attack/paradox - the likelihood of a correct guess for a key size n is $2^{(n/2)}$, so here it's 2^{64}) (Martin, pp.202, 212).

If it were a hash alone, Marvin could have used a rainbow table to reduce this work effort - which is why we suggested the DigiPound Bank use a salt to generate hash values for storing their passwords in *answer 2*.

Now let's put this theory into practice from the view of a real attacker.

Specific attacks allowed by modern infrastructure

Please turn back to Figure 2. Above are the opportunities Marvin has to take Alice's bank details and money. We could never list every exploit available – there are just too many, and the landscape changes daily. What we will do however is give an insight into 'Marvin's first thoughts' for each target above, and give the reader an appreciation of just how many exploits there are for each path (with the help of a series that this author has owned during his teens).

I. Terminal station exploit

Here Marvin seeks to gain access to Alice's, Bob's or the DigiPound Bank's computers.

How?

He installs Malware on one of the client terminals. If he gets a rootkit[19] onto Alice's machine, he can potentially steal the contents of Alice's wallet.

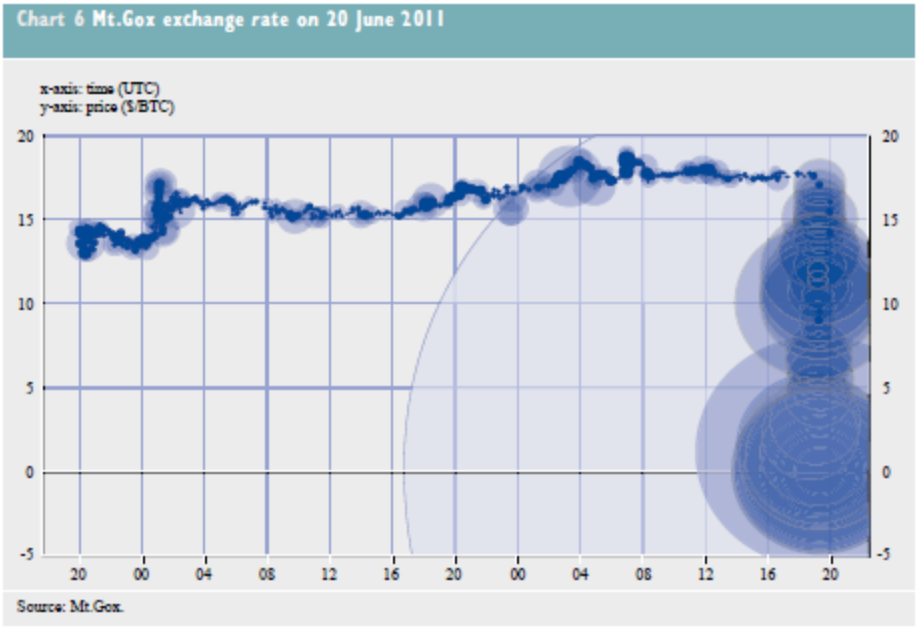
This exploit occurred with the BitCoin project, which is the most widespread alternative digital currency in existence:

"From time to time, Bitcoin is surrounded by controversy. Sometimes it is linked to its potential for becoming a suitable monetary alternative for [activities prohibited by law], as a result of the high degree of anonymity.[9] On other occasions, users have claimed to have suffered a substantial theft of Bitcoins through a Trojan that gained access to their computer.[10] The Electronic Frontier Foundation, which is an organisation that seeks to defend freedom in the digital world, decided not to accept donations in Bitcoins anymore. However, practically identical problems can also occur when using cash, thus Bitcoin can be considered to be another variety of cash, i.e. digital cash. Cash can be used for [activities prohibited by law] too; cash can also be stolen, not from a digital wallet, but from a physical one; and cash can also be used for tax evasion purposes. The question is not so much related to the format of money as such (physical or digital), but rather to the use people make of it. Nevertheless, if the use of digital money in itself complicates investigations and law enforcement, special requirements may be needed."[20]

P.T.O.

[19] "**rootkit** is a stealthy type of software, often malicious, designed to hide the existence of certain processes or programs from normal methods of detection and enable continued privileged access to a computer." Rootkits, Part 1 of 3: The Growing Threat". McAfee. (2006), source: http://en.wikipedia.org/wiki/Rootkit

[20] "Virtual Currency Schemes"(October 2012) by the European Central Bank (http://www.ecb.int/pub/pdf/other/virtualcurrencyschemes201210en.pdf) is an excellent report of modern digital currencies and the practicalities of real transaction anonymity.



Bitcoin has also featured in the news, in particular following a cyberattack perpetrated on 20 June 2011, which managed to knock the value of the currency down from USD 17.50 to USD 0.01 within minutes. Apparently, around 400,000 Bitcoins (worth almost USD 9 million) were involved. According to currency exchange Mt.Gox, one account with a lot of Bitcoins was compromised and whoever stole it (using a Hong Kong based IP to login) first sold all the Bitcoins in there, only to buy them back again immediately afterwards, with the intention of withdrawing the coins. The USD 1,000/day withdrawal limit was active for this account and the hacker was only able to exchange USD 1,000 worth of Bitcoins. Apart from this, no other accounts were compromised, and nothing was lost.[12]
Chart 6 shows the evolution of Bitcoin's exchange rate on the Mt.Gox exchange platform during the hours of the incident, and is also the expression of how an immature and illiquid currency can almost completely disappear within minutes, causing panic to thousands of users.

European Central Bank (October 2012)

He could also compromise Alice's private key by issuing a system call (we hope the OS prevents this from being exported, although there is no way of guaranteeing Alice's computer or mobile has a patched OS or up to date anti-virus to reduce the chances. Worms and Malware, such as the widespread worm 'Duqu' have been known to do this:

"According to McAfee, one of Duqu's actions is to steal digital certificates (and corresponding private keys, as used in public-key cryptography) from attacked computers to help future viruses appear as secure software. Duqu uses a 54×54 pixel jpeg file and encrypted dummy files as containers to smuggle data to its command and control center."[21]

So, a server vulnerability can be used to conduct theft, aid certificate fraud and snooping.

[21] Venere, Guilherme; Szor, Peter (18 October 2011). "The Day of the Golden Jackal – The Next Tale in the Stuxnet Files: Duqu" , http://blogs.mcafee.com/mcafee-labs/the-day-of-the-golden-jackal-%E2%80%93-further-tales-of-the-stuxnet-files

II. Web application exploit

Here Marvin seeks to gain access to Alice's, Bob's or the DigiPound Bank's computers.

How?

A web application exploit is a similar kettle-of-fish – entrance is through a website bug (this may or may not compromise the OS userland, allowing the installation of privileged software in I). This may be introduced by a coding error, or an unpatched web framework. Consider a web form at the DigiPound Bank server that reveals user account details or unused DigiPound Bank serial IDs to Marvin when he enters an unterminated SQL query (SQL injection).

Let's also take a look at the following web framework bug that caused theft from the central BitCoin exchange to steal funds in January 2013:

"All of the current versions of the Ruby on Rails Web framework have a SQL injection vulnerability that could allow an attacker to inject code into Web applications. The vulnerability is a serious one given the widespread use of the popular framework for developing Web apps, and the maintainers of Ruby on Rails have released new versions that fixes the flaw, versions 3.2.10, 3.1.9 and 3.0.18."

January 3, 2013, 10:16AM

https://threatpost.com/en_us/blogs/sql-injection-flaw-haunts-all-ruby-rails-versions-010313

Bitcoin exchange hacked via Rails exploit, funds stolen

http://www.reddit.com/r/netsec/comments/16dtf5/Bitcoin_exchange_hacked_via_rails_exploit_funds/

Forum announcement:

"Kumala (https://vircurex.com): We sadly need to announce that our wallet has been compromised thus DO NOT send any further funds to any of the coin wallets, BTC, DVC, LTC, etc. We will setup a new wallet and reset all the addresses. This will most likely take the whole weekend.

Further update: The system was not breached, no passwords were compromised (they are salted and multiple times hashed anyways). The attacker used a RubyOnRails vulnerability that was released yesterday (http://www.exploit-db.com/exploits/24019/) to withdraw the funds therefore.

Currency Exchange: https://vircurex.com"

January 11, 2013, 12:19:25 PM

"stan.distortion: Ouch, good luck with it. Bitcoin central's down too, looks like someone's [Marvin] being a [expletive]"

“Endgame: Sorry to hear that. How bad is the loss? Will users be out of pocket, or can vircurex cover it?”

“Kumala (https://vircurex.com): Before the wild speculations beginn, the service will be recovered and we pay the losses out of our own pockets”

https://Bitcointalk.org/index.php?topic=135919.0

If you replace the words “Bitcoin” and “Bitcoin central” with “DigiPound” and “DigiPound Bank” you get an idea of just how realistic such an exploit would be.

III. Protocol/network infrastructure exploit

Let's say that Marvin has gained access to Alice or Bob's computer or network. The e-wallet is too heavily encrypted to break in (BitCoin wallets use AES-256 with client-side encryption to lock down the wallet.dat file[22]). Instead Marvin wishes to intercept the protocols and decrypt the stream.

How?

He is going to use a network snooping tool, like Wireshark. His intention is to view the transferred data between any of our two parties, and hopefully decrypt it if it utilizes a weak enough cipher suite.

Let's take the magnificent proof-of-concept from security Analyst Alec Waters of the blog "Wirewatcher: Looking beyond the obvious" (http://wirewatcher.wordpress.com/2010/07/20/decrypting-ssl-traffic-with-wireshark-and-ways-to-prevent-it/).

The first piece of advice echoes what we mentioned in I: if the attacker has gained access to the private keys then decrypting traffic is trivial – it's not even an exploit.

"Protection of one's private key is at the core of any system using asymmetric keys. If your private key is compromised, the attacker can either masquerade as you or they can attempt to carry out decryption as outlined above. Keys stored in separate files like the ones above are particularly vulnerable to theft if access permissions are not set strictly enough, or if some other vulnerability allows access. Certain operating systems like Windows and Cisco's IOS will try to protect the keys on your behalf by marking them as "non-exportable". This is meant to mean that the OS won't divulge the private key to anyone under any circumstances, but clearly there comes a point where some software running on the box has to access the key in order to use it. This simple fact can sometimes be exploited to export non-exportable keys".

So again, we want to make sure that none of our party computers is compromised to run Marvin's software (I), by making sure they run authorized, uncompromised software which is always up-to-date.

He still has an opportunity to exploit if the chosen cipher suite was weak (and again, it is not really an exploit, because it cannot be patched – it is an accepted result of using legitimate tools with a specific cipher suite configuration).

"As we've seen, the RSA key exchange is susceptible to interception if one is fortunate enough to have the server's private key. By using a flavour of Diffie-Hellman for key exchange instead, we can rule out any chance of an attacker feasibly decrypting our SSL traffic even if they are in possession of the server's private key."

– and that's why we opted for a key exchange with Diffie-Hellman in *answer 3*.

[22] "BitCoin Wallet – Be Your Own Bank", https://blockchain.info/wallet/

"We've changed the cipher from RC4 to 256 bit AES – this step is just to prove that Wireshark can decrypt AES as well as RC4.

Now, to break the decryption, alter the cipher suite to use Diffie-Hellman instead of RSA for key exchange:

openssl s_server -key testkey.pem -cert testcert.pem -WWW -cipher DHE-RSA-AES256-SHA -accept 443

Wireshark is totally unable to decrypt the HTTP traffic, even though it is in possession of the server's private key.

A DH key exchange is by design resistant to eavesdropping, although can be susceptible to a man-in-the-middle attack unless both parties identify themselves with certificates."

This really highlights why both web programmers and network administrators need to choose their cipher suites wisely, and make sure their web language frameworks are well patched, and their servers are well managed[23]. A single exploit (like the Ruby on Rails exploit that afflicted the BitCoin central server above) can be exploited before a patch is even released, as shown – so sometimes watching news alerts 24/7 is not enough.

[23] Speaking as someone who is the current webserver administrator and a previous Ruby on Rails programmer of http://desaldata.com (*Tagline:"Be your own consultant"*) and http://americanwatersummit.com, this really requires two separate departments with very specific human resources. A realistic environment, particularly in the case of merchant Bob, will need to argue an ongoing management budget for this.

IV. Naive operator error leading to handshake with an untrusted party

Alice sees mobile banking as a convenience, not a threat. She is doesn't realize the danger of the words 'Untrusted Certificate', and proceeds because she is in a rush. Marvin wants to exploit this user's carefree naivety and demanding lifestyle. Her phone also hasn't been synced for updates for a while.

How?

Phishing attacks

Marvin sets up replica of the DigiPound Bank currency exchange page on his own server. He buys a legitimate certificate, but not one from a trusted DigiPound Bank Certificate authority.

He sends Alice a convincing fake e-mail issuing an exchange rate that Alice takes advantage of – but the link redirects to Bob's server.

Alice's browser doesn't warn her – she sees an SSL padlock and cannot distinguish the fake URL from a valid one. Unfortunately, Bob's webserver has just rootkitted Alice's phone for later zero-day exploitation (such as installing a Trojan to pull her private-key).

When Alice types her details into a webpage or iframe controlled by Marvin, he also takes her DigiPound Bank authentication details.

Prevention:

She must be particularly vigilant in checking the SSL certificate in her address bar, or, the bank accepting this, should implement an alert system (like the Squawking Bird browser plugin that Smith mentions for invalid SSL certificates) which the bank could deploy within Alice's supplemental 2-way attestation software. The attestation software would notice that while Marvin's server has a purchased certificate, it has not been issued by the DigiPound Bank. It should prevent any transfer of funds or attempt to sync her e-wallet with an unauthorized server's IP address.

More importantly, the bank needs to educate Alice, whose mistakes are totally understandable but not excusable. Alice needs to understand that with great convenience comes great responsibility, and potential liability as well.

Browser redirection attack

Here is another way that Marvin can get Alice to use his server without Alice noticing.

Our system could be vulnerable to a browser redirection attack because of its underlying web-based infrastructure. It's common that large banks will hire out the webserver hosting management and security implementation to totally external contracted parties (it often seems more sensible to arrange this than try to maintain a secure web platform with inhouse resources, as we actually suggested earlier). This could lead to a weakening of Alice's reaction to her being redirected towards an unknown URL (or seeing an embedded iframe asking for a password). Let's take the common example of Google - a giant's URL that everyone assumes is legitimate. For followers of its beta services, being redirected from one product name to a later incarnation is expectable behaviour. This isn't such a problem if the

page that is redirecting is HTTPS - because if the second page is also HTTPS then the two will have negotiated a handshake (e.g. http://www.gmail.com redirects to https://www.google.com/accounts). But if the original is not HTTPS (e.g. http://www.gmail.com also redirects to this target), and that Marvin, in our case takes control of http://www.DigiPoundBank.com which redirects to his server, https://www.apparenttrustedthirdparty.com which harbours Marvin's purchased SSL certificate, then Alice might not think twice. However, DigiPound Bank's supplemental attestation software could verify the destination IP address and create an alert if false.

In the first scenario Alice is communicating directly with the DigiPound Bank, who appear to be the root authority. But when she checks with Bob, the software she is using should confirm that the chain(s) of Bob's certificate end at the DigiPound Bank, or a DigiPound Bank approved Authority.

Prevention:

Make sure the webserver does not get compromised by tightening file permissions, and keeping webservers and web development frameworks patched.

When DNS servers themselves are hacked (yet the webserver hosting the code remains untouched), using the DNSSEC[24] suite (from the IEFT) will prevent redirection by adding a digital signature check between the webserver and the client.

It is worth noting that upgrading software can also introduce bugs, as the Debian Open-SSL bug of 2008 shows. The Debian project was distributing a version of OpenSSL with the line:

```
MD_Update(&m,buf,j); /* purify complains */
```

removed from: md_rand.c , a file used to generate pseudo random numbers, the foundation of nonce generation.

It was removed because the tool 'purify' would throw up warning messages.

"*Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.*" Schneier on Security[25] (2008)

[24] The DigiPound Bank webserver admins should install this.
"DNSSEC adds digital signatures to normal DNS queries, substantially reducing the risk of falling victim to man-in-the-middle attacks such as the Kaminsky exploit, which caused widespread panic in July 2008.", http://www.theregister.co.uk/2010/04/13/dnssec/

[25] http://www.schneier.com/blog/archives/2008/05/random_number_b.html. See also http://www.xkcd.com/424/

V. Naïve operator error leading to evesdropping

It's not always Alice's fault – the commercial world pushes immature technologies to make a profit. Alice has a home WiFi solution from a popular ISP. Her phone always trusts this connection when in range (without prompting Alice). Unfortunately, Marvin has set up his own WiFi network with the same name (which he discovered by 'wardriving[26]' in Alice's neigbourhood). She is in range of Marvin's department and the phone in her pocket connects to his replica network. During her lunch break she sits at the cafeteria and orders something from merchant Bob.

Marvin monitors the transaction using the snooping exploits in III.

Prevention: Alice should use a secure encryption protocol on her home WiFi network. Provided that Bob's wardriving effort is only scanning for unlocked networks (which he uses to take control of routers as well as setting up fake hotspots), Alice is safe – because her phone will not connect to his fake hotspot without the correct key. She needs to be vigilant about using unknown hotspots (e.g. a public cafeteria or long distance bus), but only *realistically if her transmission protocols, certificates or software are insecure.*

Cracking protocols:

Alice's ISP sent her a router when she signed up 9 years ago. It never broke, so she never received an upgrade. Consequently it uses the WEP protocol that was the standard at the time. This is also the tutorial she took the time to learn, and replicated to set up other WiFi devices in subsequent years.

Marvin is aware of a security update regarding the once popular WEP protocol:

both WEP-40 and WEP-104 "have been deprecated as they fail to meet their security goals." IEEE, 2004

He uses his tools to compromise her home network, crack her wireless key, and monitor her phone using his replica hotspot when she is in the vicinity.

Prevention:

Her ISP, becoming aware that her default router configuration was at risk, should have issued reconfiguration guidance to each customer, if not a replacement. This would have kept Alice's education current as well. Her bank issued attestation software might even have picked up on this, although it is unlikely because the availability of a secure internet connection protocol is the responsibility of a different software layer. What this really highlights is that encryption standards trusted by industry have always gone out of date, so re-education and an ongoing maintenance effort must always be budgeted.

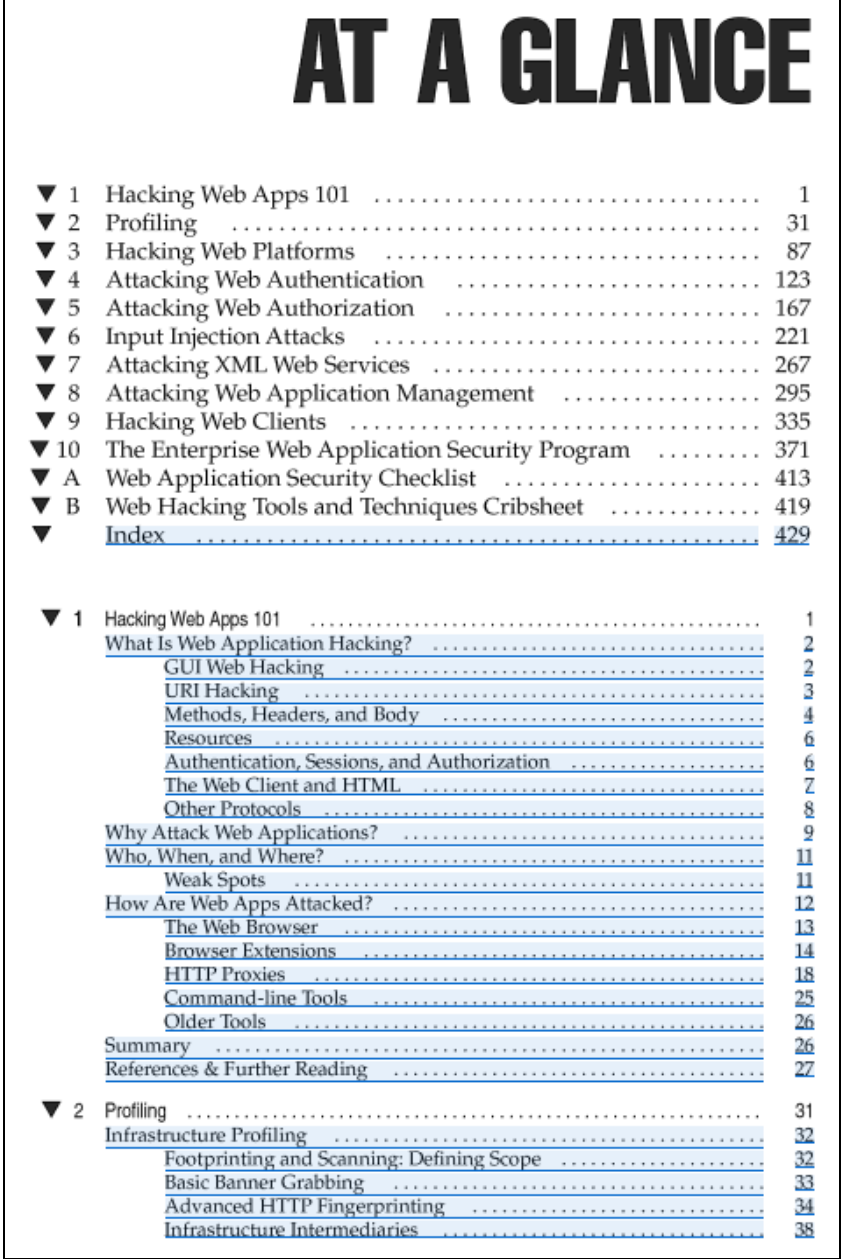
[26] "Wardriving is the act of searching for Wi-Fi wireless networks by a person in a moving vehicle, using a portable computer or PDA", en.wikipedia.org/wiki/Wardriving

Further reading

I mentioned earlier that I was unable to list every exploit in these 4 pillars of attack.

Please turn to Appendix D now for some additional external works that illustrate how each infrastructure component of our system is critical - and inevitably vulnerable.

	Certificate fraud, web application hacking



AT A GLANCE

Hacking Exposed: Web Applications (3rd edition)
J. Scambray, V. Liu, McGraw-Hill Osborne (2010)

References

[1] Kaufman, K. (2002) *Network Security: Private Communication in a Public World*, 2nd edition,Prentice Hall

[2] Schneier, B. (1995) *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd edition, John Wiley & Sons Inc.

[3] Smith, S. and Marchesini, J (2008) *The Craft of System Security*, 1st edition, Addison Wesley

[4] Martin, K. (2012) *Everyday Cryptography: Fundamental Principles & Applications*, 1st edition, Oxford University Press

Thank you to:

[3] for my first steps; easy to get into, and modern scenario overviews (peering into the cliff);

[2] for having a fundamental principle on every page; this man is a genius and the veins of the internet (diving straight in);

[1] for giving me the less drilled down content of 2: rewriting it with more present day internet applications (swimming around the bank);

[4] for really teaching me about SSL, the encryption algorithms *du jour* (performance rated) and which choices to make for this decade (getting up to a higher current platform).

All other references have been cited in the text.

Appendix A

Inspiration from my two favourite digital currencies

Name	Overview	Similarities	Useful ideas	Differences	Lessons
BitCoin	The most popular alternative digital currency system. ("Bitcoin Value". Bloomberg) Each coin is a chain of digital signatures. The owner hashes the transaction and public key of the next owner they pass the coin onto.	Isolates currency into a digital, spendable representat-ion, akin to the DigiPound.	E-wallet Digital signatures to identify sender It is too complicated to handle each coin individually, so they are grouped together per transaction (a money order). Change is sent back to the owner	Uses a peer-to-peer network with no central authority. Users mine their own coins. The P2P network addresses doublespend by timestamping every transaction, hashing them into an ongoing 'proof-of-work' chain in the coin that cannot be redone. The network uses the timestamp to check the coin was not spent before. *We* rely on the DigiPound Bank transaction log instead. Transactions are not instant (https://bitcointalk.org/index.php?topic=8143.0)– and one needs to wait for their wallet to verify the hashchain. (BitCoin magazine: http://bitcoinmagazine.com/the-mintchip-the-canadian-governments-answer-to-bitcoin/ , http://bitcoin.stackexchange.com/questions(/4020/why-does-bitcoin-make-my-computer-lag-freeze),(/1906/why-does-it-take-15-minutes-for-my-bitcoin-client-to-start/):"The downside of a decentralized system is that you cannot trust anything and must check everything yourself."	If one loses their wallet.dat file, the contents disappear forever. Heavy reliance on the P2P network – the system breaks down if the largest pool of CPU resources is controlled by an attacker.
Digicash	" Then there was DigiCash and the brilliant blind signature protocol from Dutch cryptographer David Chaum. Combining a powerful centralized issuing mint with true transaction irreversibility and anonymity, DigiCash would have flourished if it weren't for the legacy intermediaries that tend to insert themselves into fledgling centralized systems when they smell a loss of revenue. The misadventures of DigiCash paved the way for needing decentralized systems and BitCoin elevated it to marquee feature by resolving the double-spend problem through the distributed block chain." Forbes	*(As above)* Uses a central authority	E-wallet Blind signatures Degree of anonymity comparable to real cash Recipient used a central authority to verify received funds	Difficult to find text on the specifics of Digicash since it collapsed in 1998 – mainly just historical stories (http://cryptome.org/jya/digicrash.htm). We know that there were individual documents for each money denomination (pennies to dollars) – each document signed by the bank. The spender would combine them to make a payment. The recipient would verify received funds with a central authority to check if it had already been spent. However, it gave us blind signatures (which lead to fair signatures and voting protocols), which are well documented in papers, and by Kaufman & Schneier. Despite this, all the papers I read imply that it is Alice who generates the money (money orders) and has the bank sign them. In our description, we expect the DigiPound Bank to be issuing the money to Alice – I amended my solution to account for this (*answer 5*).	Being able to reveal Alice's identity if she doublespends is valuable. True transaction anonymity leads to abuse. See "Digicash and the perfect crime", Schneier, 145

Appendix B

Realworld programmers deciding on modern cipher suites

Discussion board:

http://security.stackexchange.com/questions/8343/what-key-exchange-mechanism-should-be-used-in-tls (2011)

What key exchange mechanism should be used in TLS?

There are many key exchange mechanisms that can be used in TLS. Among them are RSA, ECDH_ECDSA, ECDHE_ECDSA, ECDH_RSA, ECDHE_RSA and others. Which of these are more cryptographically secure and can be used for securing connection with web site?

web-application encryption ssl key-exchange

edited Oct 24 '11 at 18:05

asked Oct 24 '11 at 10:07

Andrey Botalov
1,052 7 25

1 Answer

You may use a key exchange (as part of a cipher suite) only if the server key type and certificate match. To see this in details, let's have a look at cipher suites defined in the TLS 1.2 specification. Each cipher suite defines the key exchange algorithm, as well as the subsequently used symmetric encryption and integrity check algorithms; we concentrate here on the key exchange part.

- RSA: the key exchange works by *encrypting* a random value (chosen by the client) with the server public key. This requires that the server public key is an RSA key, *and* that the server certificate does not prohibit encryption (mainly through the "Key Usage" certificate extension: if that extension is present, it must include the "keyAgreement" flag).
- DH_RSA: the key exchange is a *static Diffie-Hellman*: the server public key must be a Diffie-Hellman key; moreover, that certificate must have been issued by a Certification Authority which itself was using a RSA key (the CA key is the key which was used to sign the server certificate).
- DH_DSS: like DH_RSA, except that the CA used a DSA key.
- DHE_RSA: the key exchange is an *ephemeral Diffie-Hellman*: the server dynamically generates a DH public key and sends it to the client; the server also *signs* what it sends. For DHE_RSA, the server public key must be of type RSA, and its certificate must be appropriate for *signatures* (the Key Usage extension, if present, must include the digitalSignature flag).
- DHE_DSS: like DHE_RSA, except that the server key has type DSA.
- DH_anon: there is no server certificate. The server uses a Diffie-Hellman key that it may have dynamically generated. The "anon" cipher suites are vulnerable to impersonating attacks (including, but not limited to, the "Man in the Middle") since they lack any kind of server authentication. On a general basis, you shall not use an "anon" cipher suite.

Key exchange algorithms which use elliptic-curve cryptography are specified in another RFC and propose the following:

- ECDH_ECDSA: like DH_DSA, but with elliptic curves: the server public key must be an ECDH key, in a certificate issued by a CA which itself was using an ECDSA public key.
- ECDH_RSA: like ECDH_ECDSA, but the issuing CA has a RSA key.
- ECDHE_ECDSA: the server sends a dynamically generated EC Diffie-Hellman key, and signs it with its own key, which must have type ECDSA. This is equivalent to DHE_DSA, but with elliptic curves for both the Diffie-Hellman part and the signature part.
- ECDHE_RSA: like ECDHE_ECDSA, but the server public key is a RSA key, used for signing the ephemeral elliptic-curve Diffie-Hellman key.

- ECDH_anon: an "anon" cipher suite, with dynamic elliptic-curve Diffie-Hellman.

So, what shall you choose, for a Web site ? Your main constraints are:

- You want a cipher suite which is supported by most clients; in practice, this rules out elliptic curve cryptography (elliptic curves are mightily cool, but not well supported yet in the field -- consider that according to gs.statcounter, as of September 2011, 40% of client systems still use Windows XP, and almost 5% use IE 7.0).
- You want a cipher suite which is compatible with your server key type and certificate. This, in turn, depends on what the CA accepts (the CA which sold you the certificate). 99.9% of the time, this means RSA. Everybody does RSA. Diffie-Hellman keys in certificates, and DSA signatures, used to be promoted by NIST (the US federal agency which deals with such matters) because there was a patent on RSA; but that patent expired in 2000. Diffie-Hellman (as part of certificates) is specified by ANSI X9.42, a standard which is not free (so opensource free-time developers are reluctant to implement it) and not all that clear either. So nobody really uses Diffie-Hellman in certificates. DSA is more widely supported (its defining standard is free and quite readable) but not to the point of being non-anecdotic when compared to RSA.
- You do not want to use an "anon" suite because that's insecure against active attackers, and most client browsers have the "anon" suites deactivated by default.

So you choice is basically between "RSA" and "DHE_RSA". The latter may have a slightly higher computational cost, although you would need to have at least a few hundred new connections per second to actually see a difference (I insist on the "new": since TLS includes an abbreviated handshake which can build on the key exchange of a previous connection, the actual key exchange with asymmetric cryptography only occurs once per new client browser in the last minute). So, in practice, no measurable difference on the CPU load between RSA and DHE_RSA.

DHE_RSA offers something known as Perfect Forward Secrecy, a pompous name for the following property: if your server gets thoroughly hacked, to the point that the attacker obtains a copy of the server private key, then he will also be able to decrypt *past* TLS sessions (which he recorded) if these sessions used RSA, while he will not be able to do so if these sessions used DHE_RSA. In practice, if the attacker could steal your private key, then he probably could read the 10000 credit card numbers in your site database, so there is little reason why he should even bother recording and decrypting previous sessions because this would yield only a dozen extra numbers or so. PFS is mathematically elegant, but overhyped. If it still a nifty acronym and can make a great impression on the weakly-minded, as part of a well-thought public relations campaign.

Summary: use DHE_RSA.

answered Oct 24 '11 at 11:50

Thomas Pornin
70.8k 13 133 249

Could you sort elliptic-curve key exchange algoritms in terms of their relative security and give details about their browser support? Unfortunately I'm not able to find details about it. – Andrey Botalov Oct 25 '11 at 17:18

RSA with a key of 1024 bits or more, Diffie-Hellman modulo a prime of 1024 bits or more, ECDH with a curve of 160 bits or more, are all in the "can't break it with today's technology" category. Thus they are all "secure" and it is difficult to state that one is "more secure" than any other in a meaningful way. – Thomas Pornin Oct 25 '11 at 17:24

1 RSA works everywhere. DHE_RSA works everywhere too (at least since IE 5 and Netscape 4, if your archeologist skills go that deep). DHE_DSS has more limited support (I think IE 6 accepts it only when used with 3DES as symmetric encryption). For anything with elliptic curves, you could experience some success with the most recent IE and Firefox, provided that you stick to the P-256 standard elliptic curve, and none other. – Thomas Pornin Oct 25 '11

at 17:27

I disagree with you calling PFS overhyped. IMO it is an essential feature for anybody who cares about privacy.
– CodesInChaos Jun 21 '12 at 22:08

1 DHE_RSA is not supported by any version of Windows SChannel that used by IE. ECDHE_RSA is supported by Vista and later. – Yuhong Bao Sep 10 '12 at 21:43

Appendix C

Using secret splitting (or secret sharing) to reveal distributed secrets - *doublespend cheaters*

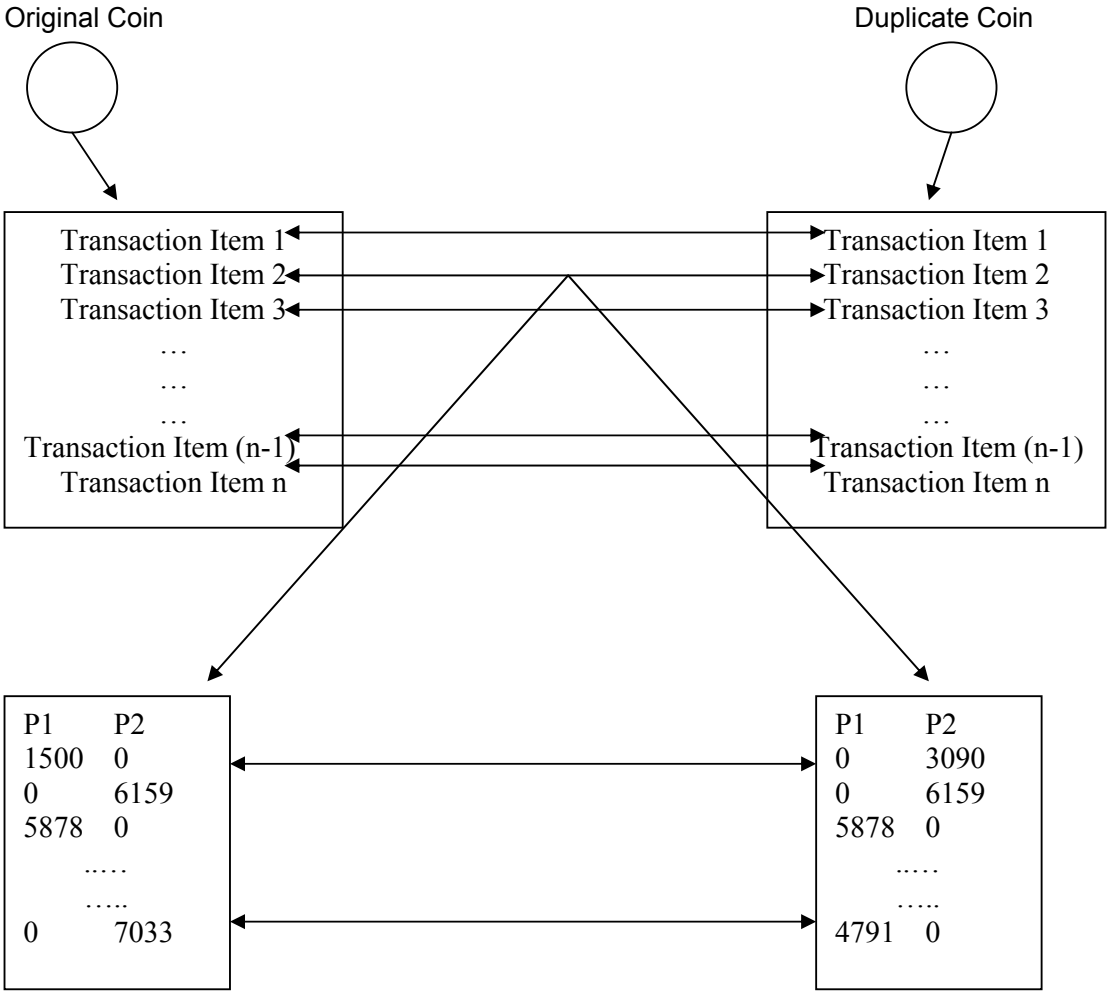
Extract from:

Digital Cash, S. Cattani, University of Birmingham (2004), p.12

http://www.cs.bham.ac.uk/~mdr/teaching/modules03/security/students/SS4/DigitalCash.pdf

How does this detect double spending?

If a user makes a copy of a coin before they spend it, they have the possibility to spend that coin again. However, when the coin is finally returned to the issuer, it will be possible to discover the culprit. This is achieved by combining a particular part of the identity from the original coin with its corresponding part from the copied coin. Note that the corresponding part will have been blanked out in the original coin. For example, let's assume user with id 2510 makes a copy of a coin and spends it twice. The diagram below shows exactly how double spending is detected:



At **transaction 2** the user made a copy of the coin and spent it elsewhere. Once the coin finally reached the issuer, they can match up pair of corresponding P1 and P2 to reveal the identity of the user who has spent the coin twice.

1500 XOR 3090 = 2510 or 7033 XOR 4791 = <u>2510</u>
(Notice that <u>2510</u> was the id of the user)

Appendix D

Further reading for attackers: Marvin’s bookshelf

Wireless network hacking, exploiting misconfiguration

AT A GLANCE

Hacking Exposed Wireless (2nd edition)
J. Cache, J. Wright, McGraw-Hill Osborne (2010)

Local network hacking, exploiting LAN misconfiguration and server OS vulnerabilities

Hacking Exposed: Network Security Secrets & Solutions (7th edition)
S. Mcclure, J. Scambray, McGraw-Hill Osborne (2012)

Exploiting OS flaws and naïve operators

Hacking Exposed: Malware and Rootkits (1st edition)
A. Davis, S. Bodmer, McGraw-Hill Osborne (2010)