# Digital Signatures

Public-key encryption is more computationally intensive than symmetric one. That’s why, it is generally used to encrypt a few small blocks of information, exchanged as part of a protocol.

Digital signature is implemented using asymmetric cryptography.

Digital signature works in two ways. It can be used by the sender, C, of a document to prove that he authored the document, and it can be used by the receiver of the document as evidence that C authored it. The latter case is referred to as nonrepudiation: once you send a signed check, you cannot at a later time, deny that you authorized a money transfer. A digital signature also serves a third, very important purpose. Once a document is signed, it cannot be modified without invalidating the signature. Hence, a digital signature guarantees data integrity.

Digital signatures use a property of many asymmetric encryption algorithms – the roles of the public PC and private RC keys are symmetric:



Hence, a simple-minded signature algorithm is one in which C signs a document, M, by encrypting it with his private key, RC. If the receiver can recover meaningful information (e.g., an ASCII string) by decrypting a message using PC, the receiver can conclude that the message could have been generated only by C, since only C knows RC. This technique assumes that the receiver knows C’s public key, PC. Note that the message is not hidden. Message hiding is not the purpose of a digital signature protocol.

As far as asymmetric encryption is time consuming, some function, f, of M is computed – generally a hash – that produces a result that is considerably smaller than M itself. f(M) is also called a message digest of M. f(M) is encrypted with RC and referred to as a digital signature, which is transmitted along with M.

Thus, C sends two items, and M, to the receiver. The receiver decrypts the first item using PC and then compares the outcome with the result of applying f to the second item. If the two are the same, the receiver should be able to conclude that M could have been generated only by C. To safely allow such a conclusion, however, we must deal with some other issues.

# Digital Signatures (Cont 1)

Consider an intruder that listens to and copies a signed transmission, (,M), from C.

1. The intruder might use the signature  to sign a different message, M’, in an attempt to fool a receiver into believing that C sent M’. The intruder can succeed in this attack if it can construct M’ such that f(M)=f(M’). To prevent this attack, f is required to be a one-way function: f has the property that, given an output, y, constructing an argument, x, such that f(x)=y is computationally infeasible. For example, a one-way message digest function might produce an output string that satisfies the following properties:
2. All values in the range of f are equally likely
3. If any bit of the message is changed, every bit in the message digest has a 50% chance of changing.

Property (a) guards against the possibility of finding an M’ such that f(M)=f(M’) , because all values are equally possible. Property (b) ensures that f(M) and f(M’) are not the same simply because M and M’ are related or similar messages.

Under these conditions, it is extremely difficult for the intruder to construct a message M’ such that f(M)=f(M’) and hence to find a message, M’, to which the signature  can be attached. Furthermore, the intruder cannot forge the signature that can be used with M’, , since he does not know RC.

1. The intruder might attempt to copy and then resend a signed message a second time, in what is referred to as a replay attack. A replay attack can be dealt with by having the signer construct a timestamp and include it in M. Since the digital signature is calculated over the entire message, the intruder cannot change the timestamp. Assuming that clocks at all sites in the network are roughly synchronized, if the receiver keeps a list of the timestamps of all recently received messages and rejects arriving messages containing timestamps in the list (recall that timestamps are globally unique), it can detect a message’s second arrival. The key point here is that a particular timestamp is never used twice. Hence, a replay attack can also be dealt with if each message has a unique sequence number.

# Digital Signatures (Cont 2)

Digital signature guarantees integrity of the message. Also, it guarantees nonrepudiation. If message contents should be kept in secret, it can be encrypted with another key.

# Key Distribution and Authentication

When communicating, parties must use keys, and, hence, keys should be distributed before session begins. Also, parties should be sure that they communicate with the party which it claims to be. For example, am I sending my credit card number to Macy or to a Macy’s impersonator? Or, should the requestor be allowed to withdraw money from a Jody’s bank account?

# The Kerberos Protocol: Tickets

Kerberos system was designed at MIT. Kerberos is a middleware module that can be incorporated into a distributed computing system.

The Kerberos protocol involves the use of an intermediary process, called a key server (or, Key distribution server, KDS). KDS creates session keys on demand and distributes them in such a way that they are known only to communicating processes. For this reason, it is referred to as a trusted third party.

Each user wishing to participate in the protocol registers a symmetric user key with the KDS. User keys are not session keys. They are used only in the key distribution protocol at the start of a session.

Assume that a client, C, wants to communicate with a server, S. C and S have previously registered user keys  and , respectively, with KDS.  is known only to C and KDS. Similarly,  is known only to S and KDS. KDS is trusted by C in the sense that C assumes that KDS will never communicate  to any other process and that data structure it uses to keep  is protected from unauthorized access. Similarly, KDS is trusted by S.

Kerberos introduces the concept of a ticket to distribute a session key. To understand the role of a ticket, consider the following sequence of steps, illustrated in Figure 26.4, which forms the heart of the protocol.

# The Kerberos Protocol: Tickets (Cont 1)

Figure 26.4. Sequence of messages used to authenticate a client in symmetric encryption

# Protocol

1. C sends to KDS a message, M1 (in the clear), requesting a ticket to be used to authenticate C to S. M1 contains the names of the intended communicants (C,S).
2. When KDS receives M1, the following takes place:

(a) KDS (randomly) constructs a session key, Ksess,C&S

(b) KDS sends to C a message, M2, containing two items:

 (i) [Ksess,C&S, S, LT]

 (ii)  [Ksess,C&S, C, LT] – the actual ticket,

 where LT is the lifetime (the time interval) over which the ticket is valid.

1. When C receives M2, it performs the following steps:
	1. C recovers Ksess,C&S from the first item using  (it cannot decrypt the ticket)
	2. C saves the ticket until it is ready to request some service

Observe that KDS does not know the actual source of M1: it could have been sent by an intruder, I, posing as C. However, KDS encrypts M2,

# Protocol (Cont 1)

making the information returned accessible only to C and S. Key  can be constructed by C using a one-way function of the user’s password PC:



Thus, only C, using PC, can construct, and hence only C can retrieve Ksess,C&S from the first item in M2.

Later, when C wants to request service from S, the following takes place:

1. C sends to S a message, M3, containing the arguments of the request (they might or might not be encrypted using Ksess,C&S), the ticket, and an authenticator (see below).

Only S can decrypt the ticket and recover the items it contains. However, the ticket (containing C) alone is not sufficient to authenticate C to S since I could have copied it in step 2 and replayed it to S with its own request. A timestamp might be useful in preventing a replay, but the timestamp cannot be stored in the ticket because the ticket is meant to be used by C multiple times during its lifetime. C therefore sends an authenticator along with its ticket. An authenticator consists of C’s name together with a current timestamp, TS, encrypted with Ksess,C&S:



and is meant to be used only once. S can decrypt the authenticator by using Ksess,C&S (which it determines by decrypting the ticket).

At this point, S knows that the ticket could have been constructed only by KDS since only KDS knows KS,KDS. Furthermore, since S trusts KDS and since each time Ksess,C&S is transmitted it is encrypted by either KC,KDS or KS,KDS, S knows that only C (and KDS) knows Ksess,C&S. The authenticator contains some plaintext (e.g., C) encrypted by Ksess,C&S, that can be compared with the contents of the ticket (which also contains C). If they match, S concludes that C must have constructed the authenticator.

However, several possible attacks must be ruled out:

1. I attempts a replay attack in which it copies both the ticket and the authenticator from M3 and uses them at a later time. To combat this, we must take it impossible for an authenticator (in contrast to a ticket) to be used more than once. A new authenticator (with a unique timestamp) is constructed by C for each of its requests. The authenticator is live if its time timestamp is within the lifetime (LT) of the accompanying ticket. To ensure that a copy of an authenticator is

# Protocol (Cont 2)

of no value, and that S can defend itself against a replay, S uses the following protocol:

* 1. If the received authenticator is not live, S rejects it
	2. S maintains a list of authenticators it has received that are still live. If the received authenticator is live, S compares it against the list and rejects it if a copy is found. By maintaining lifetime information, S can limit the number of authenticators it has keep on the list
1. I intercepts M3 (it does not reach S) and tries to use the ticket and authenticator for its own request for service. However, if C has chosen to encrypt the arguments of its request with Ksess,C&S, then I cannot substitute its own arguments because he does not know Ksess,C&S. Sending the entire intercepted message at a later time accomplishes nothing for I since it simply causes C’s original request to be serviced.
2. I intercepts M1 (it does not reach KDS) and substitutes the message (C,I). KDS responds to C with a ticket encrypted with. I’s goal in this attack is to copy the message M3 that it hopes that C will subsequently send to S. Since the ticket is encrypted with I’s private key, I can extract the session key. In this case S will not be able to decrypt the ticket, but I can determine private information about C contained in the arguments that C sends to S. The protocol defends against this attack by including the server’s name in the first item of M2, which in this case will be I instead of S. C uses this information to determine the identity of the process that can decrypt its request message.

# Single Sign-On

Kerberos provides a property, referred to as single sign-on, that is becoming important as client interactions become increasingly complex. Complex interactions frequently involve access to multiple resources and hence multiple servers. Each server needs to authenticate the client and, in the worst case, has its own interface for doing so. If the client uses the same password for all servers, security can be compromised; if the client uses different passwords, he must remember all of them. In either case he must engage in multiple authentication protocols, and the system administrator must keep the authentication information associated with each server current as client information changes.

With single sign-on, the client needs to authenticate itself only once. Kerberos provides this property by concentrating authentication in an authentication server, AS, which authenticates C at login time using the password supplied by the client. Since the identity of the servers the client intends to access might be not known at this time, it is not possible for AS to construct the appropriate tickets (since each ticket must be encrypted with a particular server’s key). Instead, it returns a ticket-granting ticket to C, which is used for requesting service from a particular server, called the ticket-granting server, TGS – also part of Kerberos. Later C can request the specific ticket it needs (for example, a ticket for S) from TGS using the ticket-granting ticket.

The authentication server generates a session key, , that C can use to communicate with TGS, and returns to C (in format similar to that of M2 in the simplified protocol):

* -  is a session key for communicating with TGS
*  - the ticket-granting ticket for TGS

where  and  are keys that C and TGS have registered with AS.

Later, when C wants to access a particular server, S, it sends a copy of the ticket-granting ticket together with the server’s name (and an authenticator) to TGS. TGS then returns to C (again in a format similar to that of M2):

*  -  is a session key for communicating with S
*  - this is a ticket for S (note that S’s private key,  is available to TGS).

# Single Sign-On (Cont 1)

C thus obtains a different ticket for each server it accesses. It engages in a single authentication protocol, and since the use of tickets is not visible at the user level, the user interface is simplified. Also, since authentication is concentrated in a single server, the administration of authentication information is simplified.

# Nonces

Suppose two processes, P1 and P2, share a session key, , and P1 sends an encrypted message, M1, to P2 and expects an encrypted reply, M2. When P1 receives M2, how can P1 be sure that it was constructed by P2? It might seem that P1 can just decrypt M2 using  and see if the result makes sense. Often, however, determining whether a string makes sense requires human intervention, and in some cases even that does not help. Consider the case in which M2 simply contains a data string (an arbitrary string of bits – perhaps the weight of some device) calculated by P2. An intruder might substitute a random string for M2. When P1 decrypts that string using , it might produce another string that looks like a data string. Unfortunately, P1 cannot determine whether or not the string is correct without repeating P2’s calculation (which it is not in a position to do). Alternatively, the intruder might replay an earlier message sent during the same session and hence encrypted with . In some cases, such a replay might be a possible correct response to M1 (two devices could have the same weight), and hence P1 is fooled into accepting it.

A nonce can be used to solve this problem. A nonce is a bit string created by one process in a way that makes it highly unlikely that another process can create the same string. For example, a randomly created bit string of sufficient length created in one process probably will not be created later by another process. Nonces have a variety of use, one of which is related to authentication.

To solve the above problem, P1 include a nonce, N, in M1, and P2 includes N+1 in M2. On receipt of M2, P1 knows that the sender must have decrypted M1, since N+1, not a simple replay of N, is returned. This implies that the sender knows  and is therefore P2.

In Kerberos, the timestamp TS (which is already part of the authenticator) can be used as a nonce so that no additional items need be added. The server can include TS+1 in M4.

# Nonces (Cont 1)

Nonces are often used in cryptographic protocols for a completely different reason. Appending a large random number to the plaintext before encrypting a message makes it considerably harder for an intruder to decrypt the message by guessing parts of its contents – for example, guessing the expiration time or some other information in a credit card – and using that information to reduce the cost of a brute-force search to discover the key. This use of a nonce is sometimes referred to as adding salt to a message. In some protocols, a nonce used for this purpose is called a confounder.

# Secure Sockets Layer Protocol: Certificates

Servers (perhaps representing business) that want to authenticate themselves to other parties as part of an Internet transaction can use a certification authority, which acts as a trusted third party.

A CA uses public-key encryption to generate certificates, which certify the association between a principal’s name (e.g., Macy) and its public key. The certificate contains (among other items) the principal’s name and public key, and it is signed with the private key of the CA. Since the CA’s public key is well known, any process in the system can determine validity of the certificate. Hence, if a client wants to communicate securely with Macy, it can encrypt a message using the public key found in a valid certificate containing the name “Macy” and be certain that only a process with knowledge of Macy’s private key will be able to decrypt the message. Certificates thus solve the problem of distributing public keys reliably, which is the key distribution problem for asymmetric encryption.

Any Internet server, S, that wants to obtain a certificate from a CA first generates a public and private key and then sends the public key, plus other information, to the CA. The CA uses various means to verify the server’s identity (perhaps by communicating with personnel at the server’s place of business by phone and ordinary mail) and then issues it a certificate containing, among other items,

* The CA’s name
* S’s name
* S’s URL
* S’s public key
* Timestamp and expiration information

The CA signs the certificate and sends the signed certificate to S; S verifies correctness of the certificate.

# SSL Protocol

The Secure Sockets Layer (SSL) protocol uses certificates to support secure communication and authentication over the Internet between a client and an Internet server (or between servers). By using certificates, SSL is able to eliminate the need for an online key server (as in Kerberos), which can be a bottleneck in transaction systems that process thousands of transactions per second.

A goal of SSL is to authenticate a server to a client. Since this is done using a certificate, each server that wants to be authenticated must first obtain a

certificate. Clients, on the other hand, are not generally registered with certification authorities and hence do not have certificates or the encryption keys associated with them. A logged-in client is typically represented by a browser, which usually does not have a private key of its own. Rather, the browser contains the public keys of all certification authorities that have made arrangements with that browser’s vendor. The browser does not actually communicate with a CA during the SSL protocol; nor does a CA know any private information about a browser.

The SSL protocol authenticates the server to the client and establishes a session key for their use. A browser engages in the SSL protocol when it connects to a server whose URL begins with https: (instead of http:), which indicates an SSL-encrypted HTTP protocol.

Assume that a browser, C, connects to a server, S, that claims to represent a particular enterprise, E (for example, Macy’s). In this case, the protocol consists of the following steps:

1. S sends C a copy of its certificate signed by the CA – in the clear
2. C validates the certificate’s signature using the CA’s public key (included in its browser) and hence knows that the public key in the certificate belongs to the enterprise named in the certificate.
3. C generates and sends to S a session key encrypted with the public key in the certificate.

Note that C, not S, generates the session key because, at this point in the protocol, C can communicate securely with S using the public key in the certificate, but S cannot communicate securely with C. Once the session key has been established, C and S can use it to exchange encrypted messages.

In many applications, S does want to ensure that it is talking to a particular client. One way to provide such authentication is for client and server to agree on a password, which the server stores and client supplies after the session key is established.

# Passport: Single Sign-On

Microsoft Passport is an Internet protocol that uses an authentication server, A, to implement single sign-on. A stores the password of each customer, C, and a symmetric encryption key, , for each server, S, that has registered for its service. It also stores a symmetric key, , for its own use. In simplified form, the protocol consists of the following steps:

1. When S wants to authenticate C, it sends a page to C’s browser that contains A’s address. The page is redirected (redirected pages are not displayed) from C to A. The effect is as if, after receiving the page, C had clicked on A’s address

Redirection can be made by, for example,

<meta http-equiv="Refresh" content="15; URL=../action/redirect.html"/> (see <http://www.w3.org/WAI/UA/TS/html401/cp0305/0305-REDIRECT.html> for more details)

1. A sends a page to C’s browser requesting C’s password
2. C enters its password and clicks the submit button. An SSL session is established between C and A, and C sends its password to A using the session key established as a part of the SSL protocol
3. A verifies that the password is correct
4. A sends a page and a cookie to C. The page states that C has been authenticated and is redirected to S. It is encrypted with , and hence S can verify that it came from A. The cookie also states that C has been authenticated. It is encrypted with  and placed on C’s browser. Its use will be explained below.
5. S sends a page to C that includes a (second) cookie to be placed on C’s browser encrypted with a key known only to S. Thus if C returns to S’s site, S can retrieve the cookie and determine that C was previously authenticated

Suppose that C later visits a different server, S’, that also offers Passport authentication, and S’ asks A to authenticate C. After step 1 of the procedure, A can retrieve the cookie it previously put in C’s browser and hence knows that C has been previously authenticated. A can now implement single sign-on. A skips steps 2 through 4 (it does not have to ask again for C’s password) and executes an abbreviated version of step 5 (it does not have to place another cookie on C’s browser).

A’s cookie is similar to a Kerberos ticket, but the password protocol does not have the extra security offered by a Kerberos authenticator and hence is subject to some of the attacks that the authenticator addresses.

# Passport: Single Sign-On (Cont 1)

For example, A leaves a cookie on C’s browser after C completes its interaction with S. If C’s interaction originates from a public terminal (for example, in public library), a subsequent user of the terminal might (perhaps inadvertently) be authenticated as C. Kerberos deals with this type of threat by requiring that the client constructs an authenticator, and this requires that the client knows information contained in the ticket (the session key, and his name). In Passport, however, the cookie can be used on behalf of a client without the client demonstrating any knowledge of its contents. To circumvent this problem, most sites have a button that C can use to remove A’s cookie from the browser.

Cookies are files saved on a client side by a server. Cookies may be used by a server to adjust appearance of sent to a client pages according to the client’s preferences.

In particular, cookies may contain authentication information. Some samples on programming database access using ASP, taken from H.M. Deitel, P.J. Deitel, T.R.Nieto, E-Business & e-Commerce. How to program. – Prentice Hall, 2001, ISBN 0-13-028419-X, Chapter 25 Active Server Pages, pp. 806-818 are given in <http://cmpe.emu.edu.tr/chefranov/Cmpe552-Fall2016/index.htm>

# Paying with a credit card

E-Wallet protocol by Microsoft uses Passport procedure to authenticate client C to both S and E-Wallet server. E-Wallet server stores user’s credit card number, mailing address, etc. Only a single password need be entered in authentication procedure. After the authentication has been completed, specified items are sent to the merchant (in encrypted form) as a part of the purchase interaction. Thus the user does not have to reenter credit card information and mailing address for each purchase with each merchant.

Verified by Visa is another protocol that provides single sign-on for paying with a credit card. In this protocol, the customer first enters a credit card number on the merchant’s Web page. The page is sent (in an SSL session) to the merchant who initiates an authentication protocol similar to the Passport protocol. The goal of the protocol is to authenticate to the merchant that the customer is authorized to use that credit card. In this case, the authentication is performed by an authentication server operated by the bank that issued the card. That server checks that the password corresponds to the credit number and that the credit card number is valid.

In both E-Wallet and Verified by Visa protocols, the merchant learns the credit card number of the customer.

Revealing the credit card number to the merchant is particularly problematic in electronic commerce because only the number, not the card itself, is needed to make a purchase. This makes it easier for a criminal to make purchases without the cardholder’s knowledge.

One simple approach to this problem is to use a trusted third party to whom the customer has already given her credit number and the merchant has already set up an account. When the customer wants to make a purchase, she tells the trusted third party to make a charge against her credit card and credit the merchant with the money. The most popular trusted third party of this type is PayPal, which currently has millions of registered users who execute hundreds of thousands of transactions per day, corresponding to billions of dollars in payments.

The PayPal protocol handles customer-to-customer (C2C) interactions. It allows one customer, C1, to send money to another customer, C2, whom she identifies to PayPal by his email address. Perhaps C1 just purchased an item from C2 on an auction site. An important requirement is that the money can be transferred using C1’s credit card without C2 seeing any of C1’s credit card information.

# Paying with a credit card (Cont 1)

C1 and C2 must have previously registered with PayPal, which maintains accounts for them. Registration is accomplished at the PayPal site by submitting SSL encrypted forms that contain, among other information, C1’s name, email address, credit card information, and a password.

To send a money to C2

1. C1 logs onto the PayPal site, authenticates herself with her password (using SSL), and requests that PayPal use her credit card account to send the money to C2, whom she identifies with an email address.
2. PayPal executes a transaction that takes money from C1’s credit card account, deposits the money in C2’s PayPal account, and sends C2 an email notifying him of the transaction. (This notification is why the protocol has been characterized, somewhat inaccurately, as “sending money by mail”.)

# The Secure Electronic Transaction Protocol: Dual Signatures

Secure Elecronic Transaction (SET) protocol is developed jointly by Visa and MasterCard. SET is particularly oriented toward customer-to-business (C2B) interactions. While SSL is a session-level security protocol, which guarantees secure communication for the duration of a session, SET is a transaction-level security protocol, which guarantees security for a purchasing transaction, including an atomic commit.

The SET protocol is quite complex, with many signatures and much cross-checking to increase overall security. Here we present a simplified version that demonstrates the mechanisms by which the credit number is hidden from the merchant and how the purchasing transaction is committed atomically.

The protocol involves two new ideas.

1. Each customer has his own certificate and hence his own public and private keys. These keys are used to provide one of the unique features of the protocol, the dual signature, which considerably increases the security of the transaction. The customer’s certificate also contains a message digest of his credit card number and its expiration date. Recall that information in the certificate is unencrypted. Hence, only the digest (not the credit card number itself) can be included. The digest is used to verify that the credit card number supplied by the customer corresponds to a card belonging to the customer.

# The Secure Electronic Transaction Protocol: Dual Signatures (Cont 1)

1. A new server, the payment gateway, G, operates on behalf of the credit card company. Thus, SET is a three-way protocol, involving the customer, the merchant, and the payment gateway, which acts as a trusted third party during the protocol and performs the commit operation at the end of the transaction.

The basic idea of the protocol is that customer C sends merchant M a two-part message: the first part contains the purchase amount and C’s credit card information encrypted with G’s public key (so that M cannot see the credit card information); the second part contains the purchase amount and the details of the purchase (but not the credit card information) encrypted with M’s public key. M then forwards the first part of the message to G, which decrypts it, approves the credit card purchase, and commits the transaction.

In one possible attack on a protocol such as this in which there is a two-part message, an intruder attaches the first part of one message to the second part of another. For example, having intercepted the messages for Joe’s and Mary’s purchases, an intruder can attach the first part of Joe’s message to the second part of Mary’s , hoping to force Joe to pay for Mary’s goods. One way to thwart this type of attack is to have M associate a unique Id with each transaction and to require that C includes it in both parts of the message. An attempt to unite the parts of different messages then becomes easily detectable. This does not solve the problem of a dishonest merchant, however, who associates the same Id with two different purchases so that the parts of the two resulting messages can be combined. A new mechanism is needed to overcome this type of problem. That mechanism is the dual signature, described next.

Before SET begins, C and M negotiate the terms of a purchase. The protocol begins with a handshake in which C and M exchange certificates and authenticate each other. C sends its certificate to M, and M sends both its certificate and G’s certificate to C, at which point C and M know each other’s and G’s public key. Then the purchase transaction begins.

1. M sends a signed message to C containing a (unique) transaction Id (which is used to guard against replay attacks). C uses the public key in M’s certificate to check the signature and hence knows that the message came from M and was not altered in transit.

# The Secure Electronic Transaction Protocol: Dual Signatures (Cont 2)

1. C sends a message to M containing two parts plus the dual signature:
2. The transaction Id, C’s credit card information, and the dollar amount of the order (but not a description of the items purchased) – encrypted with G’s public key:



1. The transaction Id, the dollar amount of the order, a description of the items purchased (but not C’s credit card information) – encrypted with M’s public key:



The dual signature has three fields:

1. The message digest, MD1, of the first part of the message:



where f is the message digest function

1. The message digest, MD2, of the second part of the message:



1. C’s signature of the concatenation of MD1 and MD2:



Thus, the complete dual signature is



and the complete message sent from C to M is .

The dual signature binds the two parts of the message. So, for example, an attempt by an intruder or M to associate  with  does not work since its message digest, MD2’, will differ from MD2. Although MD2’ can be substituted for MD2 in the dual signature,  cannot be used as the signature for MD1\*MD2’, and only C can compute the correct dual signature for the reconstructed message.

1. M decrypts the second part of the message with its private key (but it cannot decrypt the first part, which contains the credit card number). The merchant then
2. Uses the dual signature to verify that has not been altered in transit. It first computes the message digest of  and checks that it is the same as the second field of the digital signature (MD2). It then uses the public key in C’s certificate to check that the third field is the correct signature for the concatenation of the first two fields.

# The Secure Electronic Transaction Protocol: Dual Signatures (Cont 3)

1. Verifies the transaction Id, the dollar amount of the order, and the description of the items purchased

Next M sends a message to G containing two parts:

(a)  and the dual signature it received from C:



1. The transaction Id and the dollar amount of the order – signed with M’s private and encrypted with G’s public key:



The complete message sent from M to G is , together with copies of C’s and M’s certificates

1. G decrypts the message using its private key.
2. It uses the dual\_signature and the public key in C’s certificate to verify that  was prepared by C and was not altered (as in step 3a).
3. It uses the message digest of the credit card information in C’s certificate to verify the credit card number supplied in .
4. It uses M’s signature in and the public key in M’s certificate to verify that  was not altered
5. It checks that the transaction Id and the dollar amount are the same in  and  (to verify that M and C agreed on the purchase)
6. It checks that the transaction Id was never submitted before (to prevent a replay attack)
7. It does whatever is necessary to approve the credit card request

Then G returns a signed approved message to M. At this point, the transaction is committed.

1. When M receives the approved message, it knows that the transaction has committed. It sends a signed message to C: transaction complete. C then knows that transaction has committed.

Note how the protocol deals with some other attacks:

1. M cannot attempt to substitute different goods since the dual signature is over the description agreed to by C. By forwarding the dual signature to G, M has committed itself to that description
2. C cannot use , copied from a message submitted by a different customer in an attempt to get that customer to pay for C’s purchase by

# The Secure Electronic Transaction Protocol: Dual Signatures (Cont 4)

attaching it to . In that case, the dual signature does not help since it is computed by C. However,  and  will have different transaction Ids, so the transaction will be rejected by G. Also, credit card information in  and in C’s certificate will not comply.

# Goods Atomicity, Certified Delivery, and Escrow

Some Internet transactions involve the actual delivery of the purchased items. Transactions involving the purchase of downloaded software are in this category. Such transactions should be **goods atomic** in that the goods are delivered if and only if they have been paid for. In the context of the SET protocol, “paid for” means that the purchase has been approved by the payment gateway; in the context of electronic cash described later, it means that the electronic cash has been delivered to the merchant and accepted by the bank.

After C and M have agreed on the terms of a transaction, but before C sends a confirmation to M (step 2 of SET), M sends (downloads) the goods to C, encrypted with a new symmetric key,, that M has constructed for this purpose. M also sends a message digest of the encrypted goods so that C can verify that the encrypted goods were

correctly received. Note that C cannot use the goods at this point since he does not know .

The description, desc, that C sends to M (in ) includes both a specification of the goods and the message digest of the encrypted goods, signed with C’s private key. In effect, C is acknowledging that it has received (in encrypted form) the goods corresponding to the digest. As in step 2 of SET, the complete message is , this is C’s vote to commit. If, on receiving the message from C, M agrees that the description it has received from C is accurate, it constructs the message  as in step 3 of SET, but includes two additional items in :

1. 

2. The message digest of the encrypted goods signed with C’s private key, which it received from C in step 2, signed again (countersigned) with M’s private key.

M then sends the message to G (step 3 of SET).

**Goods Atomicity, Certified Delivery, and Escrow (cont. 1)**

Another issue in the delivery of goods over the Internet is **certified delivery**. A goods-atomic transaction guarantees delivery to the customer, but we would like to have the additional assurance that the right goods are delivered. How can M defend itself against a charge that the goods it sent do not meet the agreed specifications, and how can C be assured that the specific goods ordered are received? In particular, if there is a dispute between M and C about the delivered goods and that dispute is to be resolved by an arbiter, how can M and C present their respective cases to the arbiter? For example,

* Suppose that after decrypting the delivered goods, C finds that they do not meet their specifications and wants her money back. C can demonstrate to arbiter that the software does not work, but how can she show that this software is in fact the same software that M sent?
* Suppose that C is trying to cheat M, and the nonworking software she demonstrates to the arbiter is not the same software that M sent. How can M unmask this attempted fraud?

Another application requiring goods atomicity is the purchase of actual (nonelectronic) goods over the Internet from an unknown person or an auction site. The goods cannot be downloaded but must be sent by a shipping agent. One participant in the transaction might be suspicious that the other will not abide by the conditions of the purchasing agreement. How can both parties be sure that the transaction is goods atomic and that the goods will be delivered if and only if they are paid for?

One approach that comes close to meeting these requirements uses a trusted third party called an **escrow agent**. A number of companies are in the business of being escrow agents on the Internet. The basic idea is that, after the customer and the merchant have agreed on the terms of the purchase, instead of paying the merchant, the customer pays the escrow agent, which holds the money until the goods have been delivered and accepted by the customer. Only then does the escrow agent forward the payment to the merchant.

# Electronic Cash: Blind Signatures

The Internet purchasing transactions we have discussed so far use a credit card, which, along with a check, is an example of **notational money**. That is, your actual assets are represented by the balance in your bank account;

# Electronic Cash: Blind Signatures (Cont 1)

your credit card or check is a *notation* against those assets. At the time you make a purchase, you provide a notation that identifies you and the cost of

goods you are purchasing; the merchant trusts that you will abide by the purchasing agreement and eventually pay for the purchases with real money

– that there is enough money in your checking account or that you are in good standing with your credit card company. In either case, your bank balance will eventually be decremented to reflect your purchase.

In contrast to notational money, cash is backed by the government. Although it does not have intrinsic value (it is, after all, just a piece of paper) the public’s trust in the stability of the government causes it to be treated as if it had intrinsic value (i.e., as if it were gold). Thus, the merchant knows that he can deposit cash in his account or use it to purchase other goods without having to trust the customer. Cash is often referred to as **token money**. In the world of the Internet, token money is **electronic** or **digital** cash.

Token money offers the participants in a transaction certain advantages over notational money.

* Anonymity. Since the customer is not required to provide a signed record to complete a cash transaction, such a transaction can be performed anonymously. Neither the bank nor the credit card company knows the customer’s identity. By contrast, a credit card company keeps records of all customer’s purchases, and a bank has access to cancelled checks. These records might be made available at a later time to the government, to a court proceeding, or even to someone hoping to pry into an individual’s personal life.
* Small-denomination purchases. For each credit card transaction, the credit card company charges a fixed fee plus a percentage of the purchase price. Thus, credit cards are not appropriate for purchases involving only a small amount of money, yet many Internet vendors would like to charge a few cents for a page of information they supply to browsers. Small-denomination electronic cash would be useful for such transactions.

Hence, there is a need to support transactions based on electronic cash. Such transactions should satisfy the requirement of **money atomicity**: money should not be created or destroyed. However, since electronic cash is represented by a data structure in the system, there are several ways in which money atomicity might be violated. For example,

* A dishonest customer can make a copy of the data structure and use both the original and the copy

# Electronic Cash: Blind Signatures (Cont 2)

* Money can be created or destroyed if a failure occurs (e.g., a message is lost or the system crashes). For example, a customer who has sent a copy of the data structure to a merchant cannot determine whether the message was received by the merchant and so decides to reuse the data structure in a different purchase – even though the money was actually received. Alternatively, the message might not have been received, but the customer does not reuse the data structure, failing to realize that the payment has not been made.

What follows is a discussion of an electronic cash protocol designed to support the purchase of arbitrary (not necessarily electronic) goods. Goods atomicity is not a feature of this protocol; instead, the customer trusts that the merchant will send the goods after the transaction commits.

# Tokens and Redundancy Predicates

This protocol is based on the Ecash protocol (firstly proposed by D. Chaum in 1982). Cash is represented by electronic tokens of various denominations. Each token consists of a unique serial number, n, encrypted with a private key, known only to the bank. The terminology here is confusing since it is often said that, in electronic cash protocols, the bank “signs” the serial number to create the token. Here, the meaning of signing differs from that given previously, in which a signed item consists of the item followed by an encryption of its digest. Here, when we say that the bank signs a serial number, we mean that the bank encrypts the serial number with a private key, and the result is a token.

How does this scheme prevent intruders from creating counterfeit tokens? After all, a token is just a bit string of a certain length. The fact that the bit string is the encryption of a serial number provides no protection. You might think that we could test the token validity by decrypting it with the bank’s public key and examining the result. However, that key can be applied to any valid or invalid token, yielding a bit string, and we have no way of distinguishing a bit string that is a valid serial number from one that is not.

To prevent intruders from creating counterfeit tokens, we use a technique that requires serial numbers to be bit strings that have some special property that distinguishes them from arbitrary bit strings. For example, it might be required that the first half of the serial number be created at random and the second half be a scrambled form of the first one using a fixed and known scrambling function. Formally, we say that there is some well-known

# Tokens and Redundancy Predicates (Cont 1)

predicate, *valid*, called a **redundancy predicate**, such that, for all valid serial numbers, n, the predicate *valid(n)* is true.

Although it is assumed that the counterfeiter knows the redundancy predicate and the bank’s public key for decrypting tokens, she does not know the bank’s private key and so cannot produce tokens by encrypting a valid serial number. Hence, in counterfeiting tokens she faces the problem of finding a (fake) token that decrypts to a bit string that satisfies valid. The counterfeiter can use a trial-and-error technique to do this, but it is extremely unlikely that the result of decrypting an arbitrarily chosen bit string will satisfy *valid* if the bit string is long enough and if *valid* is such that the number of bit strings of that length that satisfy valid is a small percentage of the total number of bit strings of that length.

In its scheme for minting tokens, the bank keeps a set of public/private key pairs and chooses serial numbers that satisfy *valid*. It signs all serial numbers used in creating tokens of a particular denomination, j, with the same private key, , from the set and uses a different private key for each denomination. The bank does not keep a list of serial numbers of the tokens it has created, but if the numbers are large enough and the number of tokens minted at each denomination is limited, the probability that the bank will choose the same serial number twice can be made vanishingly small. The bank does keep a list of the serial numbers of all tokens deposited and therefore can reject a copy of a token that was deposited already. In this way, it can detect an attempt to use a duplicated token. Any customer or merchant can check the validity and denomination of a token by decrypting it with  and applying *valid* to the result.

# A Simple Digital Cash Protocol

If anonymity is not an issue, the customer, C, the merchant, M, and the bank, B, can use the following simple digital cash protocol.

## Creating Tokens

* 1. C authenticates himself to B and sends a message requesting to withdraw some specified amount of cash, in the form of tokens, from his account
	2. B debits C’s account and mints the requested tokens by making up a serial number,, for each token, such that is true. It then encrypts  with the private key,, corresponding to the token’s denomination j, to produce the token .
	3. B sends the tokens to C, encrypted with a session key generated for B and C’s use in the usual manner. A token cannot be sent in the clear because an intruder can copy it and spend the copy before C has a chance to spend the original. (In that case, the original token will be rejected by B when it is later deposited by C). At this point, the token-creation transaction is committed.
	4. C receives the tokens and stores them in his “electronic wallet”.

## Spending Tokens

1. When C wants to use some of his tokens to purchase goods from M, he establishes a session with M and generates a session key in the usual manner. He then sends a message to M containing a purchase order for the goods and the appropriate number of tokens, all encrypted with the session key.
2. Upon receiving the message from C, M decrypts the tokens and checks that they are valid and are sufficient to purchase the requested goods. M then sends the tokens to B, encrypted with a session key.
3. Upon receiving the message from M, B decrypts the tokens and checks that they are valid. It then checks its list of deposited tokens to ensure that the received tokens have not already been deposited. B then adds the received tokens to its list of deposited tokens, credits M’s account with the amount of the tokens, commits the transaction, and sends a *complete* message to M.
4. Upon receiving a *complete* message from B, M performs local commit actions and then sends a *complete* message (and the goods purchased) to C.

This protocol does not guarantee goods atomicity or certified delivery. M might not send the goods to C.

# An Anonymous Protocol and Blinding Functions

The simple digital cash protocol does not provide anonymity to the customer because the bank can record the serial numbers of the tokens withdrawn by the customer. When those tokens are deposited by a merchant, the bank could conclude that the merchant sold something to that customer. This exposes some information about the customer’s activities that she might prefer to keep private. To provide the anonymity, the protocol is modified so that the customer (not the bank) makes up a

**An Anonymous Protocol and Blinding Functions (Cont 1)**

serial number, *n*, that satisfies *valid(n),* scrambles it, and then submits it to the bank. The bank creates the token by signing the scrambled serial number (it does not know what the serial number is), using a private key

appropriate to the denomination being withdrawn. Such a signature is called a **blind signature**. The bank does not know the serial number, so it cannot trace it back to the customer when the token is later deposited by the merchant. When the customer receives the blinded token from the bank, it unscrambles it to obtain the token.

To implement a blind signature, the protocol uses a blinding function, *b*, (sometimes called a **commuting function**). The function *b* and related to it inverse, , have two properties:

1. Given *b(n),* it is very difficult to determine *n*
2. *b* commutes with the encryption function used by the bank involving the private denomination key . That is,



and as a result



Therefore C can recover the token from the blinded token.

## Creating Tokens

1. C creates a valid serial number, n, satisfying *valid*(n).
2. C selects a blinding function, b (known only to C), and blinds the serial number by computing b(n)
3. C authenticates himself to B and sends a message containing b(n), requesting to withdraw from his account some specified amount of cash in the form of tokens. (As in the simple digital cash protocol, the message is encrypted with a session key). Since B does not know the blinding function, it cannot determine n.
4. B signs b(n) with a private key, , appropriate to the token’s denomination, creating the blinded token . It debits C’s account accordingly and then returns the blinded token to C, again using a session key. Although B cannot check that C had selected a valid serial number (satisfying Valid(n)), C has no reason to construct an invalid number because he knows that B will debit his account by the amount of the token and that the token’s validity will be checked

**An Anonymous Protocol and Blinding Functions (Cont 2)**

when he attempts to spend it. At this point, the token creation transaction is committed.

1. C unblinds the blinded token by using to obtain , which is the requested token consisting of a signed valid serial number.

# Creating a Blinding Function

The protocol requires that C creates his own blinding function, *b*, unknown to B. This might seem a difficult task, but it is actually quite easy in the context of RSA algorithm for public key cryptography. In one scheme for doing this, C first generates a random number, *u*, that is relatively prime to the modulus *N* of the bank’s keys. Because u is relatively prime to *N*, it has a multiplicative inverse, , with respect to *N*, such that



To blind the serial number, *n*, C computes



and sends the result to B. Hence, the blinding function can be viewed simply as multiplication by a random number.

The signed result, *sr*, returned by B to C is



Obviously, . To recover the token, we use



The serial number *n* can be now obtained using.

# Spending Tokens

The protocol for spending and validating tokens involves the same steps as those in the simple protocol previously described. Fortunately, we did not assume that B kept a list of the serial numbers of the tokens it generated because, with the anonymous protocol, it does not know what these numbers are. When M submits the token to B for redemption, B simply assumes that, if the serial number satisfies valid, it earlier (blindly) signed that serial number. As before, B keeps a list of the serial numbers of tokens that have been deposited, so it will not accept the same token twice.

# Money Atomicity

The question is whether the electronic cash protocols achieve money atomicity. Money atomicity has two aspects:

* 1. Money might be created (outside of any transaction) if a process can make a copy of a token and then spend it. However, the bank will uncover this attempted fraud when it checks the serial number against its list of previously submitted tokens. Counterfeiting is another way of creating money, but, as we saw earlier, success in this is unlikely.
	2. Money might be destroyed as the result of a failure, but such problems can be generally dealt with.
1. In the token generation transaction, the bank debits the customer’s account, sends the token, and then commits, but the communication system loses the token, and it is never delivered to the customer. However, these protocols have the interesting property that, if the customer claims never to have received a token that the bank sent, the bank can simply send the customer a copy of the (blinded) token that it retrieves from its log. It means, that B should log generated by it blinded tokens. Even if the customer is dishonest and now has two copies of the token, only one can be spent.
2. In the token spending transaction, the system crashes after the customer sent the token but before she received a message that the transaction committed. The customer does not know whether the transaction committed before the crash and hence whether the token was actually spent. If she attempts to spend the token again, she might be accused of fraud. However, she can later ask the bank whether a particular token was spent (i.e., is in the list of spent tokens), but this might compromise her anonymity.

# Security in XML-Based Web Services

# XML Encryption

We consider the W3C Recommendation of December 10, 2002, which can be found at <http://www.w3.org/TR/2002/REC-xmlenc-core-20021210>

Suppose an XML document describing a purchase contains the buyer’s name and credit card information:

<Payment xmlns = “<http://...>”>

# XML Encryption (Cont 1)

 <Name> John Doe </Name>

 <CreditCard Limit=”5000” Currency = “USD”>

 <Number>1234 5678 9012 3456</Number>

 <Issuer>Bank of XY</Issuer>

 <Expiration>04/09</Expiration>

 </CreditCard>

</Payment>

While the name must be accessible to the merchant, we might want to reserve for the credit card company access to the credit card information. The result of encrypting the credit card information using XML Encryption is shown in Figure 26.7

Figure 26.7. An encrypted element within an XML document

<PaymentInfo xmlns = “<http://...>”>

 <Name> John Doe </Name>

 <EncryptedData Type =

“

[http://www.w3.org/2001/04/xmlenc#Element”](http://www.w3.org/2001/04/xmlenc#Element\”  )

xmlns=”[http://www.w3.org/2001/04/xmlenc#](http://www.w3.org/2001/04/xmlenc)”/>

<EncryptionMethod Algorithm =

“[http://www.w3.org/2001/04/xmlenc#tripledes-cbc”/](http://www.w3.org/2001/04/xmlenc#tripledes-cbc)>

 <ds:KeyInfo xmlns:ds =

“[http://www.w3.org/2000/09/xmldsig#](http://www.w3.org/2000/09/xmldsig)“>

 <ds:KeyName>keyABC</ds:KeyName>

 </ds:KeyInfo>

 <CipherData>

 <CipherValue>Zx23XAbc4..</CipherValue>

 </CipherData>

 </EncryptedData>

</PaymentInfo>

The credit card element has been replaced by an EncryptedData element. Its Type attribute indicates that an entire element within the document has been encrypted. The actual encrypted value of the credit card is in the CipherValue element. The EncryptionMethod element gives information about the method used for encryption. The value of its Algorithm attribute is a URI that identifies the encryption algorithm, in this case a Triple DES algorithm. The KeyName element gives information that can be used to identify the decryption key (perhaps the name of a file

# XML Encryption (Cont 2)

accessible to the receiver containing the key). The assumption is that the key is properly protected, and the receiver can produce the key given this

information. The EncryptionMethod and KeyName elements are both optional and can be omitted if the user is expected to know that information.

In some situations the same element might be encrypted more than once. For example, in a medical record the details of a particular malady might be accessible to the medical staff only while more general information might be accessible to the accounting department (for billing purposes) as well. To handle this, the details might be encrypted using a particular key or technique known only to the medical staff. As a result, the details are doubly encrypted and accessible only to the medical staff. For example, in Figure 26.7 we might encrypt both the Name element and the EncryptedData element and embed the result as CipherData in another EncryptedData element.

In Figure 26.7 the symmetric key used to decrypt the element is known to the receiver, and the sender simply identifies it using the KeyName child. Alternatively, the sender might transmit the key, in encrypted form, along with the element. In this case the key might be encrypted using the public key of the receiver. The EncryptedKey element defined in XML is used for this purpose.