Security and Electronic Commerce

Chapter 26

1

Security in Transaction Processing Systems

- Security is essential in many transaction processing applications
- *Authentication*
  - Is the user who he says he is?
- *Authorization*
  - What is an authenticated user allowed to do?
    - Only cashiers can write cashier's checks
    - Only faculty members can assign grades

2

Security on the Internet

- Security is particularly important on the Internet
  - Interactions are anonymous, hence authentication of servers and users is important
  - Eavesdroppers can listen to conversations
    - Credit card numbers can be stolen
  - Messages can be altered
- *Encryption* used to increase security

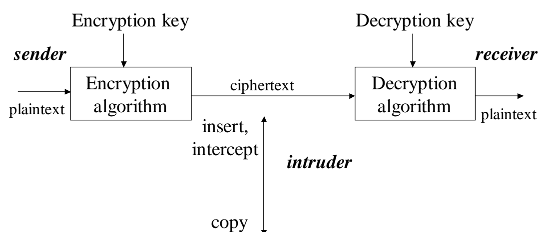
3

Encryption

- Protect information:
  - Stored in a file
  - Transmitted between sites
- Against *intruders*:
  - *Passive* intruder: eavesdrops and copies messsages
  - *Active* intruder: intercepts messages, sends modified or duplicate messages

4

Model of an Encryption System



5

Notation

- For encryption and decryption

  $ciphertext = K_{sender}[plaintext]$

  $plaintext = K_{receiver}[ciphertext]$
- then

  $plaintext = K_{receiver}[K_{sender}[plaintext]]$

6

The Encryption Algorithm

- It is assumed that the encryption algorithm is common knowledge and is known to all intruders
- The only secret is the decryption key
  - Since one approach to cracking an encryption system is to try all possible keys, the longer the key the more secure the system
- Two kinds of cryptography:
  - Symmetric cryptography

    $K_{sender} = K_{receiver}$
  - Asymmetric cryptography

    $K_{sender} \neq K_{receiver}$

7

Symmetric Cryptography

- Same key used for encryption and decryption

  $M = K[K[M]]$
- Key associated with communication *session* (not with sender or receiver)
- Computationally efficient (compared with asymmetric cryptography)
  - Hence, most security systems use symmetric techniques to encrypt data

8

Symmetric Cryptography

- *Block cipher*
  - Plaintext is divided into fixed sized blocks, which are separately encrypted
- Types of block cipher:
  - *Substitution cipher*
    - Each plaintext block is replaced by another that can be calculated using the key.

      abc → xza, def → tyy, ghi → rew, ...
  - *Transposition cipher*
    - The characters within a block are rearranged in accordance with the key (some fixed permutation):

      abc → bca, def → efd, ghi → hig, ...

9

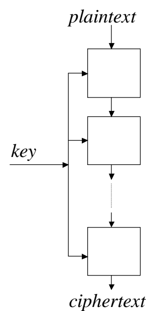
Block Cipher Attacks

- Frequency analysis attack:
  - Plaintext block frequency (calculated from a sample of normal communication) is compared with block frequency in intercepted (encrypted) message; blocks with similar frequency are matched
  - **Problem**: Frequency analysis of plaintext can be performed accurately when block size is small
    - **Solution**: use large block size
  - **Problem**: The longer the ciphertext stream, the more accurate ciphertext block frequency can be measured
    - **Solution**: change keys often

10

Data Encryption Standard (DES)

- An ANSI standard symmetric cipher widely used in commerce
- Product cipher :
  - Sequence of stages
  - Each stage is a substitution or transposition cipher
  - Block = 64 bits; key = 56 bits
    - **Problem**: Key size too small; hence “easy” to crack



11

Asymmetric (Public Key) Cryptography

- Each *user*, *U*, has a pair of related keys:

  K_u^{Pub} and K_u^{Pri}
- Different keys for encryption and decryption

  $M = K_u^{Pri}[K_u^{Pub}[M]]$
- Encryption key, K_u^{Pub}, is public knowledge
- Decryption key, K_u^{Pri}, is private (secret)
  - Anyone can send *U* a message by encrypting with K_u^{Pub}
  - Only *U* can decrypt it, using K_u^{pri}

12

Public Key Cryptography

- Current systems based on Rivest, Shamir, Adelman (RSA) algorithm
- Computationally expensive for extended exchange of data
- Often used to encrypt (short) messages of security protocols

13

The RSA Algorithm

- Pick two large random primes p and q
- Let $N = p*q$
- Pick a large integer d relatively prime to $(p-1)*(q-1)$
- Find the integer e such that $e*d = 1 \ (mod\ (p-1)*(q-1))$
- Encryption key is (e, N). If C is ciphertext, M is plaintext (a block with numerical value < N), then

  $C = M^{e} \ (mod\ N)$
- Decryption key is (d, N). To decrypt:

  $M = C^{d} \ (mod\ N)$
- Security based on the difficulty of factoring N

14

Digital Signatures

- Digital Signatures can be used for
  - Proof of authorship
  - Non-repudiation by author
  - Guarantee of message integrity
- Important for many Internet applications
- Based on public key cryptography
  - Current systems use RSA algorithm

15

Digital Signatures --Basic Idea

- Roles of public and private keys can be reversed:

  since $(M^{e})^{d} \ (mod\ N) = (M^{d})^{e} \ (mod\ N)$ it follows that

  $K^{Pub}[K^{Pri}[M]] = M$
- U encrypts message with its private key:

  $K_u^{Pri}[M]$
- Anyone can decrypt message with U's public key:

  $K_u^{Pub}[K_u^{Pri}[M]]$
  - If decryption produces an intelligible message, only U could have created it

16

Signatures and a Message Digest

- **Problem**: It is computationally expensive to encrypt an entire message with K^{Pri}
- **Solution**: Encrypt a *message digest*, $f(M)$

  $|f(M)| << |M|$
  - **Example**: f takes the hash of M
  - f is public
- Signature is $K^{Pri}[f(M)]$
- Complete signed message is $(M, K^{Pri}[f(M)])$

17

Verifying Signatures

- To verify a signed document (M, q)
  1. Compute message digest of first part, $f(M)$
  2. Decrypt second part: $K^{pub}(q)$
  3. Compare the results of (1) and (2)
- Signature is secure if:
  - f is *one-way*: Given y, it is not feasible to find an x such that $y=f(x)$
    - Hence, intruder cannot find M' to which the signature sent with M, $K^{Pri}[f(M)]$, can be attached
      - $(M', K^{Pri}[f(M)])$ is not valid
  - No *replay attack*

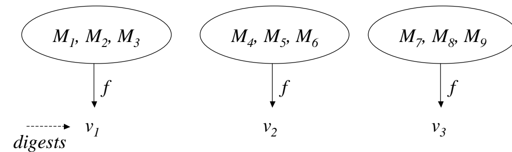
18

One-Way Function

- Over the range of possible messages, all digests are equally likely.
  - If f maps a large percentage of messages to the same digest, it may be easy to find an M' such that $f(M) = f(M')$
- If any bit of M changed, each bit of $f(M)$ has a 50% chance of being reversed
  - Guards against the possibility that closely related messages have the same digest

19

One-Way function



Sets have roughly equal size.
Elements of a set are unrelated.

20

Replay Attack

- **Problem**: Intruder copies the message and then resends it to receiver
- **Solution**: Include unique timestamp (or sequence number) in message. Receiver keeps timestamps of recently received messages and does not accept a duplicate

21

Digital Signature

- Receiver can verify who sent M
- Receiver can be sure that M has not been changed in transit (integrity)
- Sender cannot deny having sent M (non-repudiation)
- **Note**: M is sent in the clear and can be read by an intruder
  - If security it needed, M can be encrypted with another key

22

Key Distribution and Authentication

- How do two processes agree on the key(s) they will use to encrypt messages?
- How can a process be sure that it reaches agreement with the right process?
  - How does server know with which client it is communicating?
  - How can client be sure that it is communicating with intended server?
- These are problems when either symmetric or asymmetric cryptography is used.

23

Key Distribution and Authentication

- Key distribution and authentication are related and can be dealt with in the same protocol
  - You need to authenticate the process to which a key is being distributed
- Since the protocol involves the exchange of only a few messages, it can use symmetric or asymmetric techniques to encrypt protocol messages
  - Data exchange (after protocol completes) generally uses symmetric encryption
- TP monitors often provide modules that implement key distribution and authentication

24

Symmetric Key Distribution and Session Keys

- **Solution 1**: Assign symmetric key, K_P, to each process, *P*. Each communication session between *P* and another process uses K_P
- **Problem 1**: Any process that can communicate with *P* can decode *all* communication with *P*
- **Solution 2**: Session keys
  - A new symmetric key is created for each session
  - Key discarded when session completed

25

Kerberos

- Developed at MIT as middleware to be used in distributed systems
- **Goals**:
  - Authenticate a client to a server
  - Distribute a session key for subsequent data exchange between the client and the server
- Uses *symmetric* cryptography to distribute a *symmetric* session key

26

Key Server

- Kerberos uses a *key server*, *KS*: a *trusted third party* responsible for distributing keys
- Each client, *C*, and server, *S*, registers a symmetric key with *KS*
  - Client key, $K_{C,KS}$, is a one-way function of *C*'s password, PW_C
    - hence it need not be stored on the client machine
    - $K_{C,KS}$ known only to *C* and *KS*
  - Server key, $K_{S,KS}$, known only to *S* and *KS*
  - *C* and S can communicate securely with *KS*

27

Kerberos Protocol: Tickets

- (*M1*) *C* sends *(C, S)* to *KS* in the clear, asking *KS* for a ticket that *C* can use to communicate with *S*
- (*M2*) *KS* sends to *C* :

  $K_{C,\,KS}\,[K_{Sess\ C\text{-}S}\,,\ S,\ LT]$ --- *C* can decrypt this

  $K_{S,\,KS}\,[K_{Sess\ C\text{-}S}\,,\ C,\ LT]$ --- The *ticket; C* cannot decrypt this

  where:
  - $K_{Sess\ C\text{-}S}$ is a new session key created by *KS*
  - *LT* is the lifetime of the ticket

28

Kerberos Protocol: Authenticators

- When *C* receives *M2*, it
  - Decrypts first part to obtain $K_{Sess\ C\text{-}S}$
  - Saves ticket until it invokes service from *S*
- (*M3*) When *C* invokes *S* it sends:
  - Ticket
  - A newly created authenticator, $K_{Sess\ c\text{-}s}\,[C,\ TS]$
    - *TS* is a timestamp
  - Arguments of invocation encrypted with $K_{Sess\ C\text{-}S}$

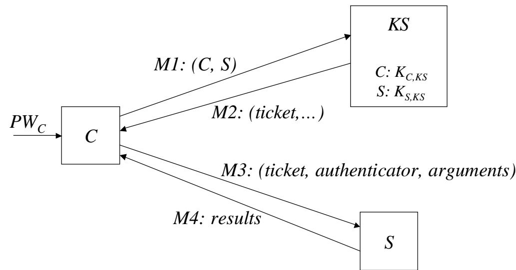
29

Kerberos Protocol

- On receiving *M3*, *S*:
  - Decrypts ticket using $K_{S,\,KS}$ to determine $K_{Sess\ C\text{-}S}$
  - Decrypts authenticator using $K_{Sess\ C\text{-}S}$
  - Checks that authenticator is *live* (*TS* is within *LT*)
  - Checks that authenticator has not been used before
    - *S* keeps a list of live authenticators that it has received
  - *C* is authenticated to *S* (*S* knows that *C* constructed M3)
- (*M4*) *S* performs requested service and returns result to *C* encrypted with $K_{Sess\ C\text{-}S}$
- Only *C* can decrypt *M4* since it is the only process (other than *S* and *KS*) that knows $K_{Sess\ C\text{-}S}$

30

The Sequence of Message in Kerberos



Possible Attacks

- Intruder, *I*, copies ticket from *M2* and tries to use it with an authenticator it creates
  - Not possible since *I* does not know $K_{Sess\ C\text{-}S}$
- *I* copies *M3* and later replays it
  - Not possible since authenticator is on *S*'s list
- *I* intercepts *M3* and uses ticket and authenticator for its own service invocation
  - Not possible if arguments encrypted with $K_{Sess\ C\text{-}S}$

Possible Attacks

- *I* obtains a ticket for *S* from *KS* and later pretends to be *C* (by sending *C* in authenticator)
  - Not possible since *I* (not *C*) is in the ticket
- *I* intercepts *M1* and sends (*C,I*) instead of (C,S); *KS* returns to *C* a ticket for *I* (instead of for *S*)
  - **Goal**: fool *C* into sending *M3* to *S* using a session key that *I* knows. *I* can copy *M3* and decrypt *C*'s arguments (note: *S* is not fooled).
  - Not possible since *I* (not *S*) is in first part of *M2*

Kerberos Protocol: Single Sign-on

- Servers often do their own authentication, maintain their own set of user passwords.
- **Problem**: A user interacting with multiple servers has to maintain multiple passwords, execute multiple authentication protocols.
- **Goal**: User supplies a single password; servers do not do authentication or keep user passwords.
- **Kerberos Solution**:
  - *C* authenticates itself once to an authentication server, *AS*
  - *C* gets a server ticket from ticket granting server, *TGS*, for each server with which it wants to interact

Kerberos Protocol: Ticket-Granting Server

- *C* sends to *AS* a request for a ticket for use with *TGS*
- *AS* sends to *C*

  $K_{C,\ AS}[K_{Sess\ C\text{-}TGS},\ TGS,\ LT]$ - session key for *TGS*

  $K_{TGS,\ AS}[K_{Sess\ C\text{-}TGS},\ C,\ LT]$ - *tkt* for service from *TGS*
- When *C* wants to invoke *S*, it sends to *TGS*:
  - *tkt*
  - An authenticator (encrypted with $K_{Sess\ C\text{-}TGS}$)
  - Arguments (*S*), (encrypted with $K_{Sess\ C\text{-}TGS}$)

Kerberos Protocol Ticket-Granting Server

- *TGS* creates a new session key, $K_{Sess\ C\text{-}S}$, and sends to *C*

  $K_{Sess,C\text{-}TGS}[K_{Sess\ C\text{-}S},\ S,\ LT]$ - session key for *S*

  $K_{S,\ AS}[K_{Sess\ C\text{-}S},\ C,\ LT]$ - ticket for *S*
- *C* and *S* then proceed as before

Nonce

- **Problem**: P_1 and P_2 share a session key, K_{Sess}. P_1 sends *M1* to P_2 and gets *M2* back.
  - How can P_1 be sure that *M2* came from P_2 and not an intruder, *I* ?
- *I* might:
  - send a random string that P_1 decrypts (using K_{sess}) to another random string that looks like a correct response
  - replay an earlier message sent by P_1 encrypted with K_{Sess} , that is a possible response (P_1 is not a server that maintains a list of timestamps)

37

Nonce

- **Solution**: Include a nonce, *N*, in *M1*
  - A random string generated by P_1
  - Long enough so that *I* cannot guess it
  - If *M2* contains *N+1* then it can only have been generated by P_2 (since only P_2 knows K_{Sess}) and it cannot be a replay

38

Authorization

- Assuming client has been authenticated, which of *S*'s operations is it allowed to perform?
  - An *access control list* stores this information at *S*
    - One entry for each user or user group
    - Entry = (user Id, access bits); each access bit corresponds to an operation that *S* exports
- Each server has an authorization policy implemented in a module called a *reference monitor* provided by TP monitor
  - Responsible for constructing, retrieving, and interpreting access control lists

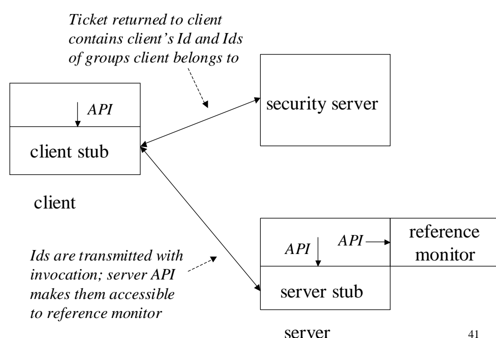
39

Authenticated RPC

- Implement authentication in the rpc stubs
- When a client wants to access a server, it invokes the client stub
- Authentication and key exchange are performed by the stub and the *security server* (*e.g.,* Kerberos)
- Security server participates in authorization by managing Ids for users and user groups and storing the Ids of each group to which a user belongs

40

Authenticated RPC



41

Internet Commerce

- Security particularly important on Internet
  - Authentication
    - Because impersonation is easy
    - We are now interested in authenticating the server to the client as well as the client to the server
  - Encryption
    - Because eavesdropping is easy
- A higher level of suspicion exists on Internet
  - Interactions are not face-to-face
  - Easy to make impressive looking Web sites

42

Secure Sockets Layer Protocol (SSL)

- Developed by Netscape for use on Internet
- Used for authentication of a server to a client (represented by a browser) and the distribution of a session key
  - Are you really sending your credit card number to Macy's?
- Server uses a *certificate* to authenticate itself

43

Certificates

- A server, *S*, registers with a *certification authority* (*CA*)
  - *CA* is a trusted third party
  - To create a certificate, *S* gives to *CA* its name, its URL, and its public key (among other items)
  - *CA* uses a number of means to satisfy itself that the party that requested the certificate is, in fact, who it claims to be
  - *CA* generates a *certificate* for *S*

44

Certificates

- A certificate contains (among other items) *S*'s name, URL, and public key (unencrypted)
- *CA* signs the certificate and sends it to *S*
  - *CA* has certified the correctness of the association between *S*'s name, public key, and URL by its signature on the certificate
  - *CA*'s public key is well known
    - A browser stores the public keys of the *CA*s that it trusts
- *S* can then distribute copies of the certificate to clients
  - Client can be sure that the public key in the certificate corresponds to the server named in the certificate
  - Solves the key distribution problem in the asymmetric case

45

Kerberos Compared with Certification Authority

- Both are trusted third parties;
- Kerberos
  - Distributes symmetric keys
  - Operates on-line, when interaction takes place since it creates a new symmetric key for each session
- Certification authority
  - Distributes public keys
  - Operates off-line, prior to interaction since public key is fixed
    - once certificate created, intervention by CA no longer required

46

Secure Socket Layer Protocol--SSL

(1) A browser, *C*, connects to a server, *S*, which claims to be some enterprise (Macy's)

(2) *S* sends *C* a copy of its certificate -- in the clear

47

SSL Protocol

(3) *C* verifies that the certificate is valid using *CA*'s public key (stored in its browser)
- *C* now knows *S*'s public key
- *C* generates a (symmetric) session key, K_{Sess} , and sends it to *S* encrypted with *S*'s public key
  - *C* generates K_{Sess} since it can send an encrypted message to *S*, but not the other way around

(4) The session follows using K_{Sess}
- SSL is a session-oriented protocol

48

Why SSL Works

- C knows it has established a session key with the enterprise that S claimed to be
  - C made up the session key and sent it to S using the public key found in its certificate
  - The certificate guarantees that the public key corresponds to the enterprise named in the certificate

49

Authenticating the Client

- If C needs to be authenticated to S, it sends its password, encrypted with the session key
  - In some applications, C might have a certificate
- In many purchasing applications, client authentication is not required
  - C sends its credit card number, encrypted with the session key
  - S learns C's credit card number (a possibly undesirable side effect)

50

Purchasing Over the Internet

- **Issue 1**: Single sign-on
  - Customer, C, interacts with several servers, S, and has to be authenticated at each
  - Microsoft Passport addresses this problem
    - Uses an authentication server (an on-line trusted third party), A
    - C and S register with A
      - A stores C's password
      - A stores a symmetric key, $K_{S,A}$, that it shares with S

51

Passport

When S wants to authenticate C:

1. S sends a page to C's browser containing A's address and attribute http-equiv="refresh" which causes the page to be redirected to A
2. A sends a page and its certificate to C requesting password
3. C sets up an SSL session to A, sends password
4. A validates password; sends page and cookie to C
   - Cookie encrypted with A's private key and stored on C's browser
   - Page contains authentication information about C encrypted with $K_{S,A}$ and is redirected to S. C is now authenticated to S

52

Passport

- Suppose C later contacts another server, S'
  - S' redirects a page through C to A requesting authentication
  - A retrieves its cookie from C's browser, indicating that C has already been authenticated
    - C does not have to resubmit its password
  - A redirects a page through C to S' indicating that C has been authenticated.

53

Passport

- **Advantages**
  - Single sign-on
  - Servers can off-load authentication
- **Disadvantage**
  - Security flaw: intruder can steal cookie off C's and use it

54

Purchasing Over the Internet

- **Issue 2**: Revealing your credit card number to the merchant
  - This is more of a problem than with normal credit card purchases since the physical card is not required
  - PayPal addresses this problem
    - Uses an authentication server (on-line trusted third party), *PP*
    - *C* and *S* register with *PP*
      - *C* stores its credit card #, password, etc., at *PP*
      - *S* maintains an account at *PP*

55

PayPal

- *C*'s "add to shopping cart" request off *S*'s web page is forwarded by *S* to *PP*
- *PP* sends its certificate to *C* and an SSL connection between them is established.
- *PP* sends a page to *C* describing the purchase for confirmation
- *C* replies to *PP* with confirmation and password
- *PP* executes a transaction that charges *C*'s credit card and credits *S*'s account

56

Secure Electronic Transactions Protocol -- SET

- A transaction-oriented protocol
- Developed by Visa and MasterCard
- The merchant, *M*, does not learn the customer's credit card number
- In addition to *C* and *M*, there is a trusted third party, *G*, the *payment gateway*
- Uses a linear commit

57

SET Protocol: The Basic Idea

- Prior to start of protocol
  - *C* sends *M* its certificate
  - *M* sends *C* its certificate and *G*'s certificate
- *C* sends *M* a message with two parts:
  - The purchase amount and *C*'s credit card information encrypted with *G*'s public key
    - *M* cannot decrypt and learn *C*'s credit card number
  - The purchase amount and the description of the item encrypted with *M*'s public key

58

SET Protocol: The Basic Idea

- *M* sends to *G* a message with two parts:
  - The first part of the message sent by *C*
  - The purchase amount of the order encrypted with *G*'s public key
- *G* :
  - Decrypts the messages (and compares amounts)
  - Approves the credit card purchase
  - Commits the transaction

59

(Simplified) SET Protocol

- Two new ideas:
  - *C*'s certificate contains a *message digest of* credit card information (in addition to other data describing *C*)
    - Credit card information itself not included
  - Security is enhanced using a *dual signature*, based on a message digest function, *f()*

60

(Simplified) SET Protocol

- (1) *M* sends *C* a message with a unique transaction identifier, *Trans_id* .
- (2) *C* sends *M*

  m_1: K_G^{Pub}*[Trans_id, credit_card_info, $_amount]*

  m_2: K_M^{Pub}*[Trans_id, $_amount, desc]*

  $f(m_1)$, $f(m_2)$, $K_C^{Pri}[f(f(m_1)*f(m_2))]$

  Dual signature

61

Dual Signature

- Dual signature verifies that:
  - The message has not been altered
    - *M* computes $f(m_1)$ and $f(m_2)$ and compares the result with the corresponding fields in the dual signature
    - *M* uses the public key in *C*'s certificate to verify that the third field is the correct signature for the concatenation of the first two fields
  - The message was constructed by *C*
  - Although the two parts are separate and encrypted in different ways, they belong to the same transaction
- *M* cannot decrypt m_1, but it can decrypt m_2

62

(Simplified) SET Protocol

- (3) *M* sends *G*

  m_1

  dual_signature

  m_4: K_G^{Pub}*[Trans_id, $_amount,* K_M^{Pri}*[f(Trans_id, $_amount)]]*

63

Dual Signature

- When *G* receives *M*'s message it uses the dual signature -- $f(m_1)$, $f(m_2)$, $K_C^{Pri}[f(f(m_1)*f(m_2))]$ -- to verify that m_1 was prepared by *C*:
  - It computes $f(m_1)$ and compares the result with the corresponding field in the dual signature
  - It uses the public key in *C*'s certificate to verify that the third field corresponds to the concatenation of the first two fields
    - It does not need m_2 to do this, since the signature contains $f(m_2)$ and the encryption is on a digest of $f(m_2)$

64

(Simplified) SET Protocol

- (4) *G* decrypts both parts of message and :
  - Uses the message digest of the credit card number in *C*'s certificate to verify the credit card number in m_1
  - Uses the signature in m_4 and the public key in *M*'s certificate to verify that m_4 was prepared by *M*
  - Matches purchase price and *Trans_Id* in m_1 and m_4
  - Checks that *Trans_id* was not used before
  - Approves the credit card debit and commits
  - Sends a commit message to *M*
- (5) *M* sends a commit message to *C*

65

Atomic Commit for SET

- SET uses a linear commit protocol
- The messages from *C* to *M* and from *M* to *G* are *ready* messages
- *G* commits the transaction
- The messages from *G* to *M* and from *M* to *C* are *commit* messages

66

Goods Atomicity

- Some Internet transactions involve the actual delivery of goods (*e.g.*, software)
- *Goods Atomicity*: The goods are delivered if and only if the transaction commits
  - Difficult to implement because the action of delivering the goods cannot be rolled back

67

Certified Delivery

- *Certified Delivery*:
  - Suppose *C* and *M* have a dispute about the delivered goods and go to an arbiter
    - If C is not satisfied with the goods, how can it prove that the goods it demonstrates to the arbiter are the goods that were delivered?
    - If *C* attempts to deceive the arbiter by demonstrating different goods than were delivered, how does *M* prove to the arbiter that *C* is cheating?

68

SET with Goods Atomicity and Certified Delivery

- SET can be enhanced to provide goods atomicity and certified delivery
- In Step (1) of the SET protocol, *M* sends *C* the goods, encrypted with a new symmetric key, $K_{C,M}$, and a message digest of the encrypted goods
  - *C* can verify that the encrypted goods were correctly received using the message digest

69

SET with Goods Atomicity and Certified Delivery

- In Step (2), *C* sends *M* the message digest of the delivered encrypted goods signed with *C*'s private key
- In Step (3), *M* verifies the message digest and sends *G*
  - The key, $K_{C,M}$
  - The message digest signed with *C*'s private key and countersigned with *M*'s private key

70

SET with Goods Atomicity and Certified Delivery

- After *G* commits the transaction in Step (5) and sends *M* the commit message, *M* sends *C* a commit message in Step (6), including the key, $K_{C,M}$
- If *M* does not send the key. *C* can get the key from *G*, which is a trusted third party.

71

SET with Goods Atomicity and Certified Delivery

- Guarantees goods atomicity
  - *C* gets the key and can decrypt the goods if and only if the transaction commits
    - If a failure occurs before the commit, the money has not been transferred and *C* does not have $K_{C,M}$
    - If a failure occurs after the commit, but before *C* gets the key, *G* has a durable copy of the key, which it can send to *C*

72

SET with Goods Atomicity and Certified Delivery

- Guarantees Certified Delivery
  - *G* has
    - The message digest of the encrypted goods signed by both *C* and *M*
    - The key, $K_{C,M}$
  - Given a copy of the goods, the arbiter can determine its correctness
    - *M* cannot deny sending it
    - *C* cannot deny receiving it

73

Escrow Agent

- A trusted third party that provides goods atomicity for non-electronic goods
  - Purchased on the Internet from someone you do not know --- perhaps at an auction site
  - Goods are delivered, not downloaded

74

Escrow Agent

- Customer, *C*, sends money to escrow agent, *E*
- *E* notifies merchant, *M* (commit)
- *M* sends goods using shipping method that allows tracking
- When *C* gets and inspects goods, he notifies *E*, which pays merchant
- If *C* gets goods (as can be demonstrated by tracking) but does not notify *E*, agent pays *M*

75

Electronic Cash

- SET involved the transfer of *notational money*.
  - Examples: credit card, checks
- *Digital money* (E-cash) has certain advantages :
  - Anonymity:
    - The merchant does not know who the customer is
    - The bank does not know with what merchant the customer is doing business
  - Small denomination purchases possible
    - Credit company charges preclude charging small purchases

76

Money Atomicity

- *Money atomicity* is a crucial requirement:
  - Money cannot be created or destroyed
    - Money might be created if someone makes an electronic copy
    - Money might be destroyed if the system fails

77

Tokens

- E-cash is represented by *tokens* of various denominations
- Each token consists of a unique *s*-bit serial number, *n*, encrypted with a private key known only to the bank $K_j^{pri}[n]$
  - The jth denomination uses the key K_j^{pri}
  - The corresponding public key, K_j^{pub}, is available to all

78

Tokens

- The number n satisfies a redundancy predicate $r()$, known to all
  - For all valid serial numbers, n, the predicate $r(n)$ is true
  - $r()$ must be such that for a randomly selected bit string p, it is extremely unlikely that $r(p)$ is true
    - Total number of serial numbers $<<<< 2^s$

Properties of Tokens

- Anyone can determine that a given bit string, t, is a valid token of a given denomination
  - Decrypt t with K_j^{pub} to obtain n
  - Verify that $r(n)$ is true
- Tokens cannot be easily counterfeited
  - If counterfeiter picks a random number t_1, the probability that $K_j^{pub}[t_1]$ will satisfy $r()$ is vanishingly small

Minting and Depositing Tokens

- Tokens are minted by the bank, *B*.
  - *B* does not keep a list of the serial numbers it has used (the likelihood of using the same number twice is vanishingly small)
- Spent tokens are returned to *B* for deposit
  - *B* keeps a list, L_S , of the serial numbers of the tokens that have been deposited
  - Using this list, *B* can reject a token that is being deposited for a second time

Simple E-Cash Protocol

- Principals are the customer, *C*, the bank, *B*, and the merchant, *M*
- Creating Tokens
  - (1) *C* authenticates herself to *B* and sends a message asking to withdraw some cash in the form of tokens from her account
  - (2) *B*
    - Debits *C*'s account
    - Mints the tokens
    - Encrypts the tokens for transmission, and sends them to *C*
    - Commits the transaction

Simple E-Cash Protocol

- Spending Tokens
  - (1) *C* sends *M* a purchase order and some tokens
  - (2) *M*
    - Verifies that the tokens are valid using K_j^{pub} and r
    - Authenticates itself to *B*, encrypts the tokens for transmission, and sends them to *B*

Simple E-Cash Protocol

- Spending Tokens
  - (3) *B*
    - Verifies that each token is valid using K_j^{pub} and r
    - Checks that each token is not in L_S
    - If all tokens are not in L_S,
      - Adds the tokens to L_S
      - Credits *M*'s account with the amount of the tokens
      - Commits the transaction and notifies *M*

Anonymous E-Cash Protocol

- Simple E-Cash protocol is not anonymous
  - When token is minted, *B* can associate *C* with the serial numbers it creates; when token is spent *B* can associate serial number with *M*
- To achieve anonymity:
  - *C* (not *B*) makes up the serial number *n* such that $r(n)$ is true
  - *B* creates the token by signing *n*, *without knowing what n is*
    - A *blind signature*

85

Blind Signatures

- The implementation of blind signatures uses the concept of a *blinding function*, *b,* and its inverse, b^{-1}, such that
  - Given $b(n)$, it is very difficult to determine *n*
  - For any private key K^{Pri}, and any *n*, $b(n)$ commutes with K^{Pri}

    $K^{Pri}[b(n)] = b(K^{Pri}[n])$

86

Anonymous E-Cash Protocol

- Creating Tokens:
  - (1) *C*
    - Selects a valid serial number *n*, such that $r(n)$
    - Selects a blinding function *b* (*known only to C*) and computes $b(n)$
    - Sends $b(n)$ to *B* and requests *B* to debit her account and mint the tokens
- It is not in *C*'s interest to cheat by picking an *n* that does not satisfy $r(n)$
  - Her account will be debited to pay for the token
  - If token not valid, it cannot be spent

87

Anonymous E-Cash Protocol

- Note that *B* cannot determine *n* since it does not know b^{-1}
  - Not a problem: even in the simple E-cash protocol, *B* did not keep a list of used serial numbers
- (2) *B*
  - Debits *C*'s account by the requested amount
  - Signs $b(n)$ with the appropriate key for the requested denomination K_j^{Pri}, creating K_j^{Pr}
  - Encrypts $K_j^{Pri}[b(n)]$ for transmission and sends it to *C*
  - Commits the transaction

88

Anonymous E-Cash Protocol

- (3) *C* unblinds the token
  - Applies the inverse blinding function, $b^{-1}()$, to $K_j^{Pri}[b(n)]$ to obtain the token $K_j^{Pri}[n]$

    $b^{-1}(K_j^{Pri}[b(n)]) = b^{-1}(b(K_j^{Pri}[(n)])) = K_j^{Pri}[n]$

89

A Blinding Function for the RSA Protocol

- C picks a random number *u* that is relatively prime to *N*
- Because *u* is relatively prime to *N*, it has a multiplicative inverse, u^{-1}

  $u*u^{-1} \equiv 1 \ (mod\ N)$
- To blind a serial number *n*, *C* computes

  $K_j^{Pub}[u] * n \ (mod\ N)$
- The signed result returned by *B* to *C*

  $sr = K_j^{Pri}[K_j^{Pub}[u]*n]$
- To unblind the signed result, *C* computes

  $K_j^{Pri}[n] = u^{-1} * sr \ (mod\ N)$

90

Anonymous E-Cash Protocol

- Spending Tokens
  - Same as before
- Protocol is anonymous
  - *B* cannot associate *C* with the serial number deposited by *M*

91

Money Atomicity in the Anonymous E-Cash Protocol

- Money might be created if a token could be copied and spent twice
  - Prevented by *B*'s list, L_s
- Money might be lost on system failure.
  - *B* logs tokens created so if *C* does not receive token, it can be resent
    - If *C* tries to cheat by saying it has not received a token it had received and *B* resends the token, *C* cannot spend both tokens
  - *C* and *M* keep copies of the tokens they send. If they do not get acknowledgements, they can ask *B* if the token was spent
    - Might lose anonymity

92

Web Services Security

XML Encryption, XML Signature
and
WS-Security

93

Why WS-Security?

- Standard signature and encryption techniques can be used to sign and encrypt an XML document but …
  - these techniques are generally tied to transmission (e.g., SSL) and do not protect the document once it arrives.
  - a document needs to be sent as a whole, and different parts might have different security requirements.
    - Transmission system cannot be expected to respect these differences
    - Example: Merchant needs to know customer's name and address, but not credit card number.

94

Complexity of the Problem

- An XML document might contain data describing an entire interaction; however each portion should be viewed only by a particular audience
  - Personal details of a medical record should not be available to a researcher, doctor should be able to see medical details but not credit card data, some medical details should not be available to administrator.
  - Different parts of document might have to be signed by different participants
  - The subsets might intersect, so multiple encryption might be required for certain portions
- Should tags be encrypted?
  - If yes, searching with XPath might be inhibited and security might be compromised (since the plaintext associated with encrypted data can be guessed)

95

WS-Security

- A standard set of SOAP extensions that can be used to implement a variety of security models and encryption techniques.
  - Supports:
    - Security token (passwords, keys, certificates) transmission
    - Message integrity
    - Message encryption
  - Makes use of other standards: XML Signature, XML Encryption

96

XML Encryption

- Example:

```
<payment xmlns="…">
   <name> John Doe </name>
   <creditCard  type="visa"  limit="5000" \>
      <number> 1234 5678 9012 3456 </number>
      <issuer> Bank of XY </issuer>
      <expiration> 04/09 </expi9797ration>
   </creditCard>
</payment>
```


XML Encryption

- Example: encrypt the credit card element (including tags)
  - Encrypted element replaces element

```
<payment xmlns="…">
  <name> John Doe </name>
  <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element
                  xmlns="…XML encryption namespace…">
    <EncryptionMethod Algorithm="…" />
    <KeyInfo xmlns="…">
       <KeyName> keyABC </KeyName>
    </KeyInfo>
    <CipherData>
       <CipherValue> AB12VY54321X ….. </CipherValue>
    </CipherData>
  </EncryptedData>
</payment>
```

encrypting an element

encryption algorithm

identify key to receiver

encrypted data

XML Encryption

- Type – granularity of encryption
  - An entire document, or an element (with or without tags) can be encrypted.
  - Different parts can be encrypted with different keys
- Algorithm –algorithm used to encrypt data
  - Example – DES, RSA
- KeyName – key is known to receiver; just identify it
- CipherData – octet stream
- The standard provides a number of options that can be used to accommodate a variety of needs

XML Encryption – Some Alternatives

1. Symmetrically encrypt data, assume the receiver knows the key and include key name (previous example)
2. Symmetrically encrypt data, include encrypted key in message (encrypted with public key of receiver) (next example)

XML Encryption and SOAP

- Store encryption key in header, encrypted data in body, in an element within body, or in an attachment
- The result of the encryption must be a valid SOAP envelope
  - Cannot encrypt *<s:Envelope>*, *<s:Header>* or *<s:Body>* elements; only their descendants

XML Encryption

Encrypted key is stored in header

WS-Security used to attach XML Encryption

wsse – prefix for WS-Security
xenc – prefix for XML Encryption
ds – prefix for KeyInfo element

```
<s:Header>
  <wsse:Security>
    <xenc:EncryptedKey >
      <xenc:EncryptionMethod
           Algorithm="…pub. key algo. to encrypt symmetric key…"/>
      <ds:KeyInfo>  <ds:KeyName> Bill  </ds:KeyName>
      </ds:KeyInfo>
      <xenc:CipherData>
         <xenc:CipherValue>abcd456…</xenc:CipherValue>
      </xenc:CipherData>
      <xenc:ReferenceList>
         <xenc:DataReference URI="#EncrData" />
      </xenc:ReferenceList>
    </xenc:EncryptedKey>
  </wsse:Security>
</s:Header>
```

optional, receiver may know it

Bill's publ. key encrypts sym. key

encrypted symmetric key

list of data items encrypted with symmetric key

XML Encryption

Encrypted data is stored in body

```
<s:Body>
  <xenc:EncryptedData Id="EncrData"
        Type="http://www.w3.org/2001/04/xmlenc#Element />
    <xenc:EncryptionMethod
        Algorithm="...symmetric algo. to encrypt data..." />
    <xenc:CipherData>
      <xenc:CipherValue>A341BB...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</s:Body>
```

identifies data

data encrypted with symmetric key

103

XML Signature

- An entire document or individual elements can be signed. Allows for the fact that
  - Different individuals might be responsible for different parts of the message
  - Some parts of the message should not be changed, others are changeable
- The signature is computed in two stages
  - A digest, using dig_fn_1, is computed of the data and encapsulated in a *<SignedInfo>* element
  - A digest, using dig_fn_2, is computed of the *<SignedInfo>* element and signed using the private key of the sender

104

XML Signature

```
<Signature xmlns=" ...XML Signature namespace..." >
  <SignedInfo>
    <CanonicalizationMethod Algorithm=" ... " />
    <SignatureMethod Algorithm="...hash/public key encryption ..." />
    <Reference URI="...locate item to be signed ..." />
      <DigestMethod Algorithm=" ...hash algorithm for item..." />
      <DigestValue>xyT14Rst...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>xYzu2fR....</SignatureValue>
</Signature>
```

digest of data

signature of entire <SignedInfo> element

105

Canonicalization Method

- **Problem**: Blank spaces, tabs, line delimiters etc. do not affect the semantics of an XML element, but two different semantically identical elements will have different digests and hence different signatures
- **Solution**: Put element into a canonical form before digesting it (but send the original).

106

Canonicalization

- **New Problem**: Receiver must know to canonicalize the data before checking the signature.
- This is one example of a transformation that the receiver must perform before digesting the data
  - Other examples: Sender might compress, encrypt, … after signing

107

Transforms

- **Solution**: Signature contains a *<Transforms>* element whose children enumerate the transformations applied to the data by the sender.
  - **Example**: Receiver must decrypt and then canonicalize the data before checking the signature.

108

Two-Stage Signature Computation

- Signature is over <*SignedInfo*> element (not over the data directly)
  - Change to data produces change to its <*DigestValue*> which produces change to signature of <*SignedInfo*>
    - Double digesting does not effect integrity of signature
  - Technique used to do the signing (but not the signature itself) is signed.
    - Defends against an attack in which intruder attempts to substitute weaker signature algorithm

109

KeyInfo Element

- **Problem**: Suppose the public key corresponding to the private key used to sign <*SignedInfo*> is not known to the receiver.

```
<SignedInfo>
    <CanonicalizationMethod  Algorithm=" … "/>
    <SignatureMethod  Algorithm="…hash/public key encryption …" />
    ….. other children …
</SignedInfo>
<SignatureValue> …. </SignatureValue>
<KeyInfo>  …. </KeyInfo>
```

produced by algorithm using a private key

identifies the private key:
- a name
- a certificate
- a corresponding public key

110

KeyInfo Element

- **Problem**: Since <*KeyInfo*> is not contained in <*SignedInfo*> it is not bound by signature to <*SignedInfo*>
  - Intruder might substitute a different <*KeyInfo*> element
- Solution: use multiple <*Reference*> elements

111

Multiple Reference Elements

```
<s:Envelope>
  <s:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          ….
          <ds:Reference URI="#mess"> … </ds:Reference>
          <ds:Reference URI="#K"> … </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue> … </ds:SignatureValue>
        <ds:KeyInfo  Id="K"> … </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </s:Header>
  <s:Body  Id="mess">
    ….
  </s:Body>
</s:Envelope>
```

part of WS-Security

both Body and KeyInfo are signed

each Reference element contains digest of item referred to

112

WS-Security

- Defines Security header block as a mechanism for attaching security-related information to a SOAP message in a standard way.
  - Uses the concept of a **security token**:
    - Asserts a claim by the sender of security-related information
      - username, PW, Kerberos ticket, key
  - Provides a mechanism for referring to security related information that is not in message
  - Tokens are children of Security header block
  - Leverages XML Encryption and XML Signature

113

Security Tokens

1. Username token element

```
<UsernameToken  Id="…">
   <Username> …. </Username>
   <Password> …. </Password>
</UsernameToken>
```

2. Binary security token – an element that carries binary security information

```
<BinarySecurityToken
   ValueType="…."          -- type of token (e.g., certificate, ticket)
   EncodingType="…." >     -- encoding format
      NmgT446C7….          -- token
</BinarySecurutyToken>
```

114

Security Tokens

3. Security token reference – a mechanism for referencing tokens not contained in the message

```
<SecurityTokenReference  Id="…" >
   <Reference  URI="…" />
</SecurityTokenReference>
```

4. *<KeyInfo>* (part of XML Signature) provides an alternate (more general) mechanism for transmitting information of this type. It can be inserted as a child of Security header block

115

Example

```
<s:Header>
  <wsse:Security>
    <wsse:BinarySecurityToken
        ValueType="…certificate…"  Id="X509Token">  xDee45TsYU….
    </wsse:BinarySecurityToken>
    <ds:Signature>
      <ds:SignedInfo>
        <ds:CanonicalizationMethod ……/>   <ds:SignatureMethod …../>
        <ds:Reference  URI="#B">                         -- body is signed
           <ds:DigestMethod …./> <ds:DigestValue …./>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue> afdSkK… </ds:SignatureValue>      -- signature
      <ds:KeyInfo>
        <wsse:SecurityTokenReference> <wsse:Reference  URI="#X509Token"/>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
  </wsse:Security>
</s:Header>
<s:Body Id="B">  …body… </s:Body>
```

WS-Security header block

XML Signature

token

information about key used in the signature is found here

116

Security Token

5. Signature – An XML Signature element can be a child of a Security header block
   - There can be multiple signatures referencing different (perhaps overlapping) components of the message
     - Example:
       - Client signs orderId header block and body of message and sends to order processing dept
       - Order processing dept adds a shippingId header block and signs it and the orderId header block and sends to billing …

117

Encryption in WS-Security

- WS-Security uses XML Encryption in a standard way to encrypt portions of a message

```
<s:Header>
  <wsse:Security>
    <xenc:ReferenceList>
      <xenc:DataReference  URI="#bodyId" />
    </xenc:ReferenceList>
  <wsse:Security>
</s:Header>
<s:Body>
  <xenc:EncryptedData  Id="bodyId" >
    <ds:KeyInfo>
      <ds:KeyName> xyz </ds:KeyName>
    </ds:KeyInfo>
    <xenc:CipherData>  <xenc:CipherValue> … </xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</s:Body>
```

ReferenceList used as a stand-alone header block; lists encrypted items

each EncryptedData element in ReferenceList provides its own key info

xyz is the name associated with the symmetric key used to encrypt data

118

Security Assertion Markup Language

(SAML)

119

SAML Goals

- **Create** trusted security statements
  - **Example**: Bill's address is *xxx@yyyyyyy* and he was authenticated using a password
  - **Example**: Bill has permission to access resource *X*
- **Exchange** security statements
  - **Example**: implement single-sign-on (SSO)
    - Bill is authenticated at his company, then wants to purchase tickets at Travel.com. He shouldn't have to re-authenticate
- SAML **non**-goal:
  - Performing authentication
  - Granting Bill access to *X*

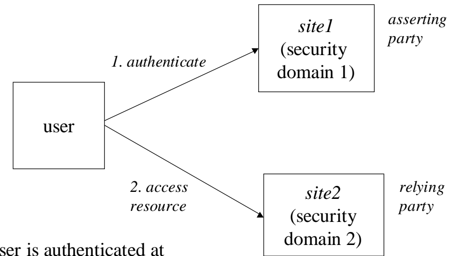
120

Why SAML?

- Permissions management data is currently handled in mostly proprietary ways, among tightly coupled modules in a single security domain.
- Web is loosely coupled, consisting of many security domains. A standard is needed to govern the transfer of assertions between domains.

121

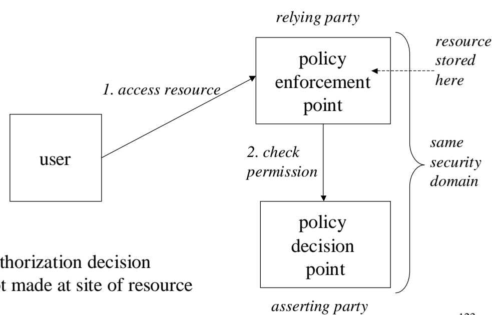
SAML Use Case: Single Sign On



user is authenticated at *site1*; then accesses a resource at *site2*

122

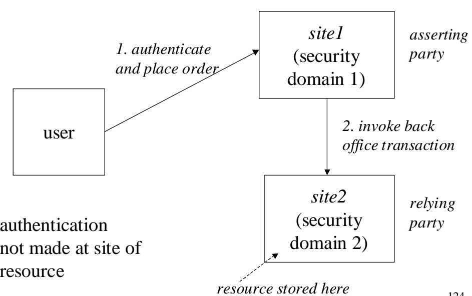
SAML Use Case: Authorization



authorization decision not made at site of resource

123

SAML Use Case: Back Office Transaction



authentication not made at site of resource

124

Why SAML?

- Cookies do not do everything SAML does
  - Cookie (signed with server's private key) can be used for re-authentication at a particular server, but is of no use at a different server
- Cross domain authentication currently requires proprietary single-sign-on software
- SAML intended as a Web standard that will supercede proprietary software

125

Security Context

- SAML must be used in the context of a trust relationship between asserting and relying parties
  - **Example**: statement "Bill has access to resource *X*" might be of no use unless we know that Bill is at the other end of the line
- Trust relationship is established using a security framework (e.g., SSL, signatures, encryption, etc.)
  - **Example:**
    - Relying party sets up an SSL connection to asserting party
      - Relying party knows (and trusts) who it is connected to (trust relationship)
    - Asserting party sends an encrypted assertion to relying party over the connection
    - Relying party can use the assertion with confidence
- Security framework is not part of SAML

126

Assertion

- A set of statements (claims) made by a SAML authority (asserting party)
  - **Authentication statement**: subject was authenticated using a particular technique at a particular time
  - **Attribute statement**: particular attribute values are associated with the subject
  - **Authorization decision statement**: subject is authorized to perform certain actions

127

Assertion

```
<saml:Assertion  xmlns:saml="…."
      … version information goes here…
       AssertionID="…."
       IssueInstant="….">
    <saml:Issuer> www.acompany.com </saml:Issuer>
    <ds:Signature>  … XML Signature goes here …   </ds:Signature>
    <saml:Subject>
         <saml:NameIdentifier  ….>  uid=joe  </saml:NameIdentifier>
    </saml:Subject>
    <saml:Conditions  …. />
     … SAML statements go here …
</saml:Assertion>
```

SAML authority making the claim

entity about which the claim is being made

128

Signature

- A signed assertion supports
  - Assertion integrity
  - Authentication of *creator of assertion* (the SAML authority)
- A signed protocol request/response message supports
  - Message integrity
  - Authentication of *message origin* (asserting party) (might be different from creator)
- A signature is not always needed
  - Assertion might *inherit* signature of containing message
  - Assertion might be received over a secure channel whose other end was authenticated by other means
- Signature is a restricted version of XML Signature

129

Subject

- Identifies the entity to which the assertion pertains
- Identifies confirmation method and (optionally) confirmation data
  - If the relying party performs the specified authentication method (perhaps using the data), then it can treat the entity presenting the assertion as the entity that the SAML authority associates with the name identifier
  - Example: method = public key, data = key information

130

Conditions

- Restrictions under which the assertion is to be used
  - **NotBefore** – earliest time at which assertion is valid
  - **NotOnOrAfter** – latest time at which assertion is valid
  - **AudienceRestrictionCondition** – assertion is addressed to a particular audience
  - **DoNotCacheCondition** – assertion must be used immediately
  - **ProxyRestrictionCondition** – limitation that the asserting party places on a relying party that wishes to create its own assertion based on this assertion

131

Authentication Statement

```
<saml:AuthenticationStatement
        AuthenticationMethod="password"
        AuthenticationInstant="…." />
```

- Asserts that the enclosing assertions' subject was authenticated by a particular means at a particular time
  - Authentication itself is *not* part of SAML
  - Statement refers to an authentication act that took place at a prior time

132

Attribute Statement

```
<saml:AttributeStatement>
   <saml:Attribute  Name =“attrib”>
      <saml:AttributeValue> val </saml:AttributeValue>
   </saml:Attribute>
</saml:AttributeStatement>
```

- Asserts that the enclosing assertion’s subject is associated with attribute *attrib* with value *val*.
  - Example: the value of the attribute *Department* associated with the assertion’s subject is *Accounting*

133

Authorization Decision Statement

```
<saml:AuthorizationDecisionStatement  Decision=“permit”
                  Resource=“... some URI ... >
   <saml:Action> Execute </saml:Action>
</saml:AuthorizationDecisionStatement>
```

- Asserts that the enclosing assertion’s subject’s request for a particular action at the specified resource has resulted in the specified decision

134

SAML Protocols

- Using a request/response pattern, SAML defines protocols/messages that
  - Request an assertion identified by unique Id
  - Request assertions containing authentication statements about the subject
  - Request assertions containing attribute statements concerning a particular attribute relating to the subject
  - Request assertions containing authorization decision statements concerning a particular resource and subject
  - Request that an authentication assertion of a particular type be created (this might involve execution of an authentication protocol)
  - Transmit protocol message by reference (artifact protocol)

135

Profiles

- SAML defines message exchange patterns that illustrate how SAML assertions can be exchanged to achieve particular goals in a particular context
  - Involve the use of SAML protocols

136

Browser/Artifact Profile

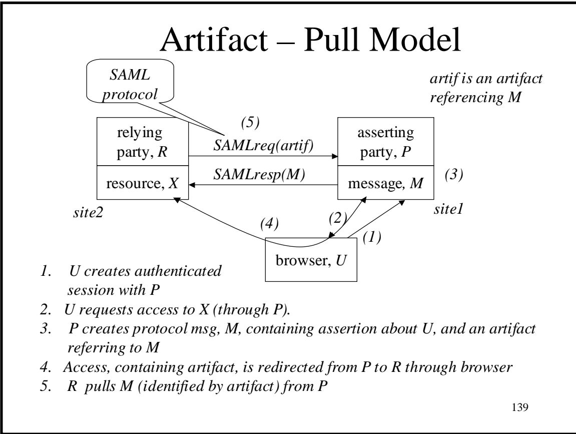
- Browser, authenticated at *site1* (asserting party) requests access to a resource at *site2* (relying party).
  - *site1* creates a protocol message containing an authentication statement and a reference to that message called an **artifact**
  - *site2* pulls the protocol message from *site1* using the artifact

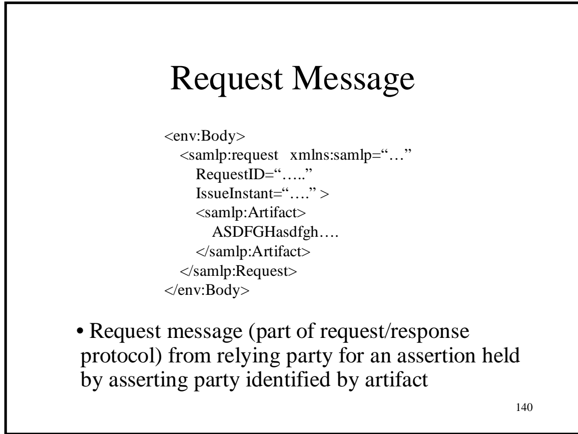
137

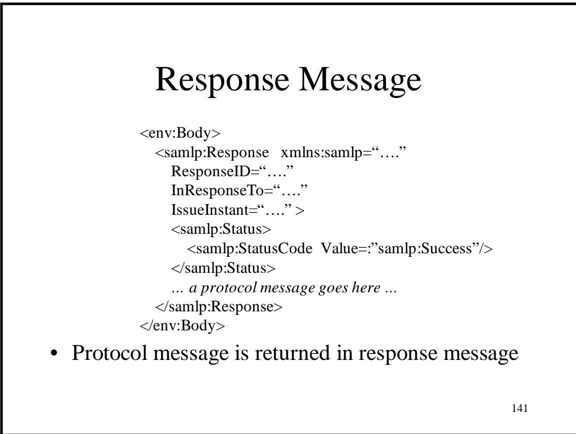
Artifact

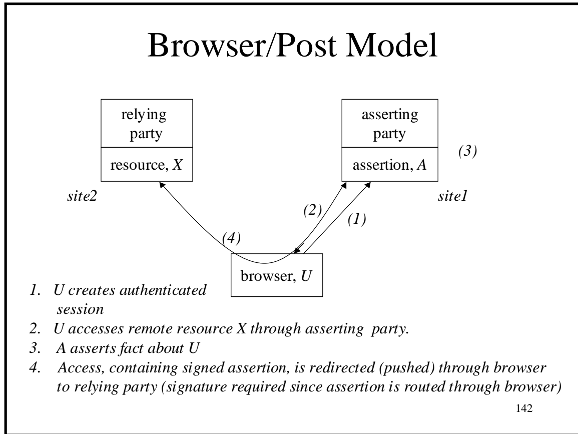
- A string consisting of
  - Identity of source site (asserting party)
  - Reference to a protocol message at source site
- Use: relying party wants to retrieve assertions in a protocol message at the asserting party; supplies an artifact that identifies the message

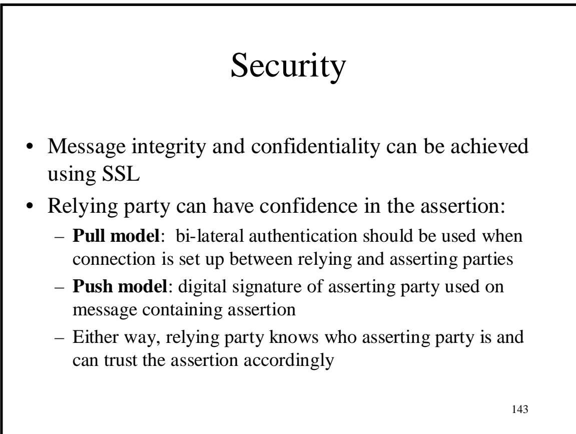
138


Artifact – Pull Model
SAML protocol
artif is an artifact referencing M
(5)
relying party, R
SAMLreq(artif)
asserting party, P
resource, X
SAMLresp(M)
message, M
(3)
site2
site1
(4)
(2)
(1)
browser, U
1. U creates authenticated session with P
2. U requests access to X (through P).
3. P creates protocol msg, M, containing assertion about U, and an artifact referring to M
4. Access, containing artifact, is redirected from P to R through browser
5. R pulls M (identified by artifact) from P
139


Request Message
<env:Body>
<samlp:request xmlns:samlp="…"
RequestID="….."
IssueInstant="…." >
<samlp:Artifact>
ASDFGHasdfgh….
</samlp:Artifact>
</samlp:Request>
</env:Body>
• Request message (part of request/response protocol) from relying party for an assertion held by asserting party identified by artifact
140


Response Message
<env:Body>
<samlp:Response xmlns:samlp="…."
ResponseID="…."
InResponseTo="…."
IssueInstant="…." >
<samlp:Status>
<samlp:StatusCode Value=:"samlp:Success"/>
</samlp:Status>
… a protocol message goes here …
</samlp:Response>
</env:Body>
• Protocol message is returned in response message
141


Browser/Post Model
relying party
resource, X
asserting party
assertion, A
(3)
site2
site1
(2)
(1)
(4)
browser, U
1. U creates authenticated session
2. U accesses remote resource X through asserting party.
3. A asserts fact about U
4. Access, containing signed assertion, is redirected (pushed) through browser to relying party (signature required since assertion is routed through browser)
142


Security
• Message integrity and confidentiality can be achieved using SSL
• Relying party can have confidence in the assertion:
– Pull model: bi-lateral authentication should be used when connection is set up between relying and asserting parties
– Push model: digital signature of asserting party used on message containing assertion
– Either way, relying party knows who asserting party is and can trust the assertion accordingly
143