Network Security I

Chapter goals:

- understand principles of network security:
  - cryptography and its *many* uses beyond "confidentiality"
  - authentication
  - message integrity
- Secure Socket Programming

Network Security I

1. *What is network security?*
2. Principles of cryptography
3. Message integrity, authentication
4. Secure socket programming

What is network security?

confidentiality: only sender, intended receiver should "understand" message contents

- sender encrypts message
- receiver decrypts message

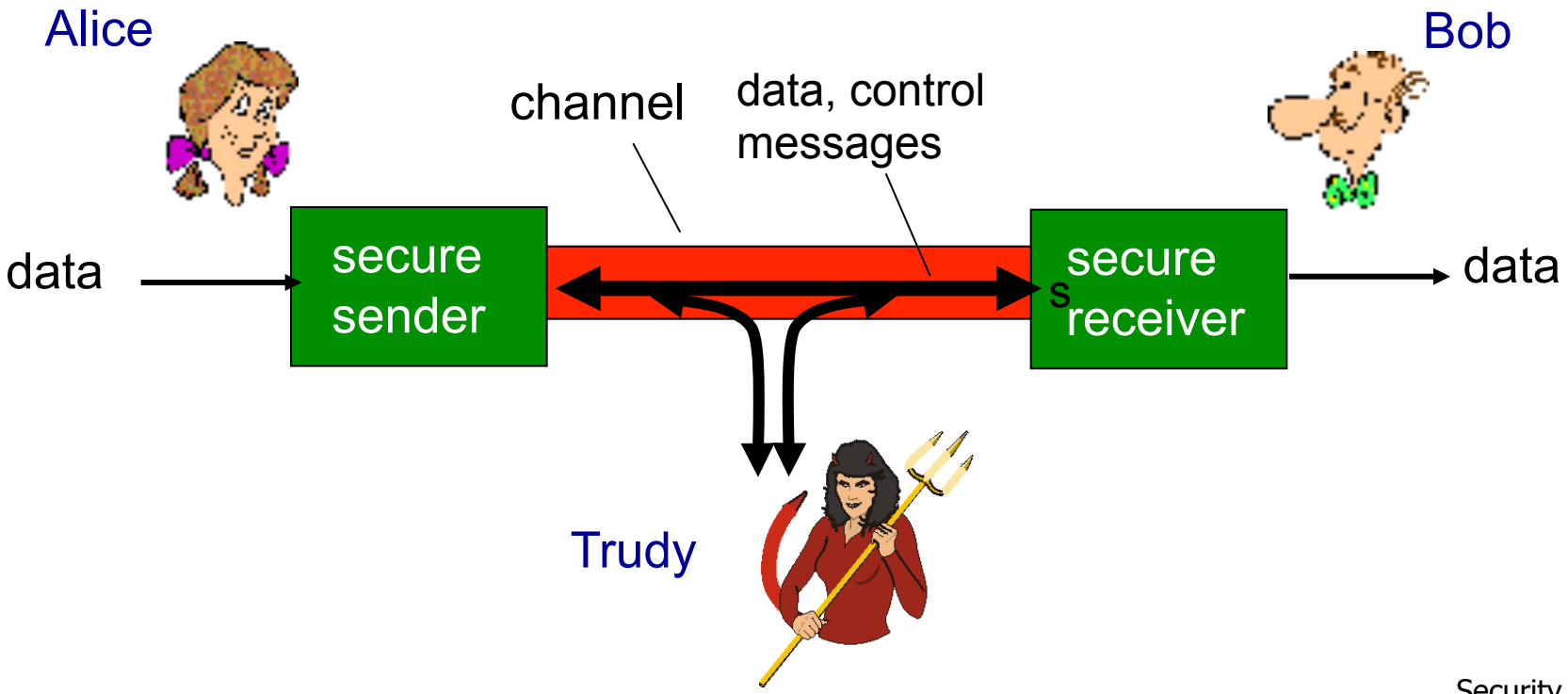
authentication: sender, receiver want to confirm identity of each other

message integrity: sender, receiver want to ensure message not altered (in transit, or afterwards) without detection

access and availability: services must be accessible and available to users

Friends and enemies: Alice, Bob, Trudy

- Bob, Alice (lovers!) want to communicate "securely"
- Trudy (intruder) may intercept, delete, add messages



Who might Bob, Alice be?

- Web browser/server for electronic transactions (e.g., on-line purchases)
- on-line banking client/server
- DNS servers
- routers exchanging routing table updates
- other examples?

There are bad guys (and girls) out there!

Q: What can a "bad guy" do?

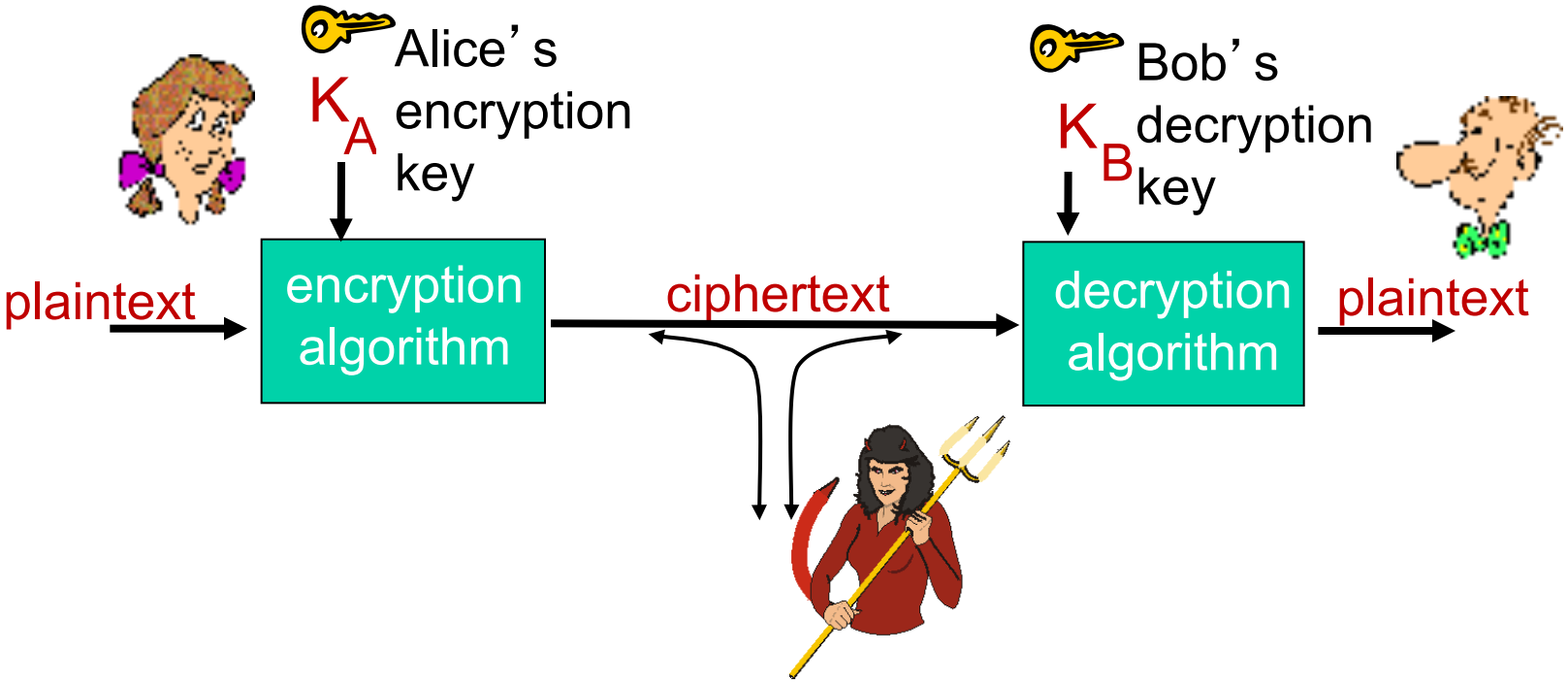
A: A lot!

- *eavesdrop:* intercept messages
- actively *insert* messages into connection
- *impersonation:* can fake (spoof) source address in packet (or any field in packet)
- *hijacking:* "take over" ongoing connection by removing sender or receiver, inserting himself in place
- *denial of service:* prevent service from being used by others (e.g., by overloading resources)

Network Security I

1. What is network security?
2. *Principles of cryptography*
3. Message integrity, authentication
4. Secure socket programming

The language of cryptography

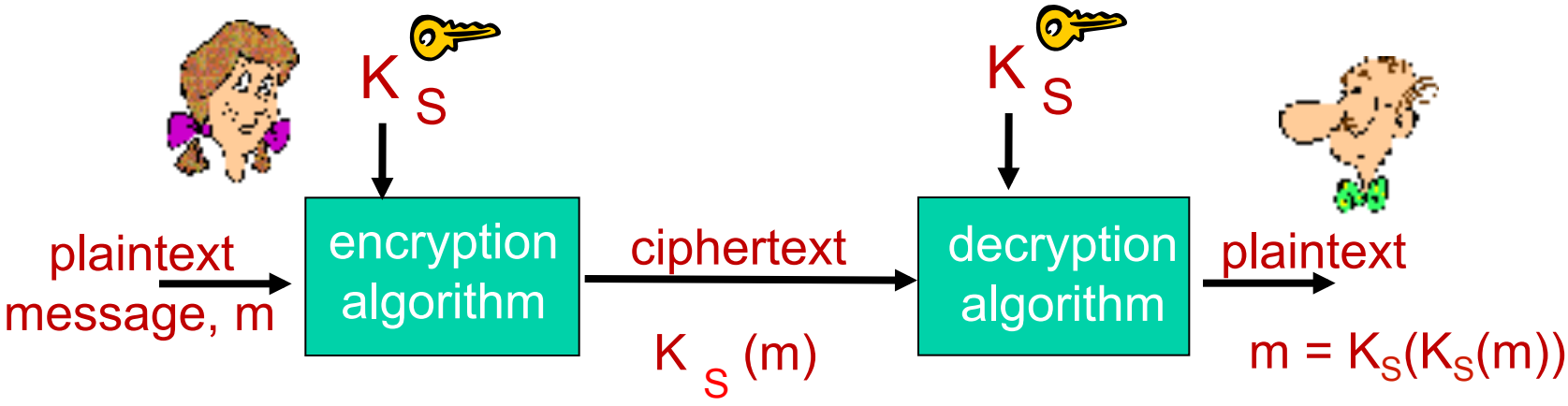


m plaintext message

$K_A(m)$ ciphertext, encrypted with key K_A

$m = K_B(K_A(m))$

Symmetric key cryptography



symmetric key crypto: Bob and Alice share same (symmetric) key: K_S

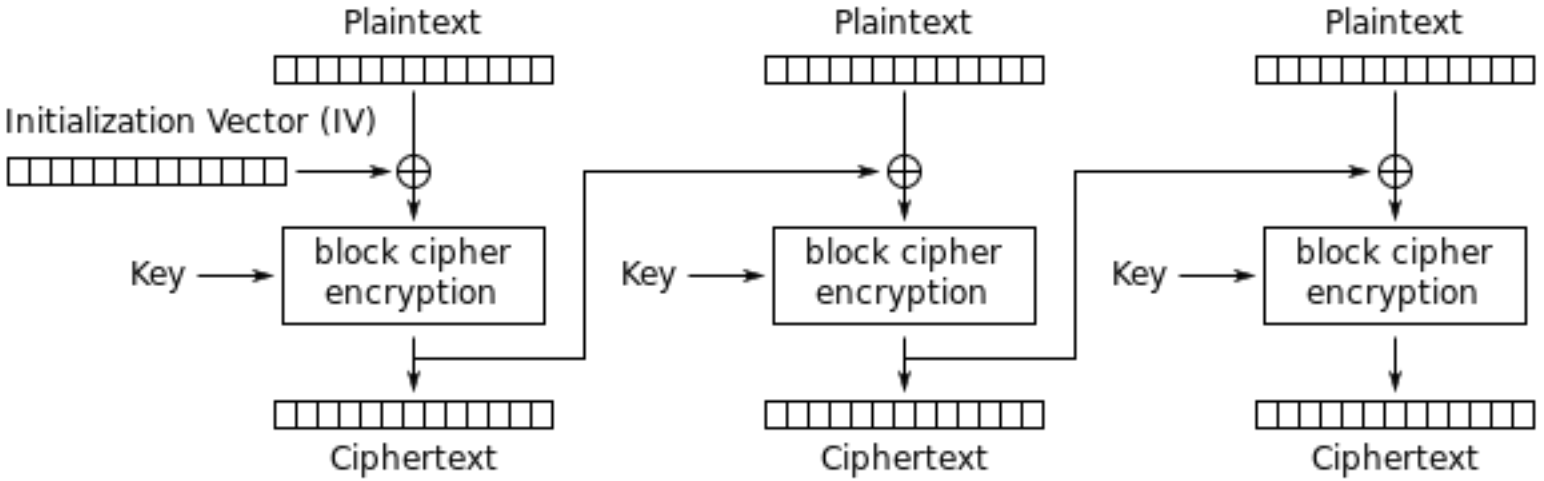
Symmetric key crypto: DES

DES: Data Encryption Standard

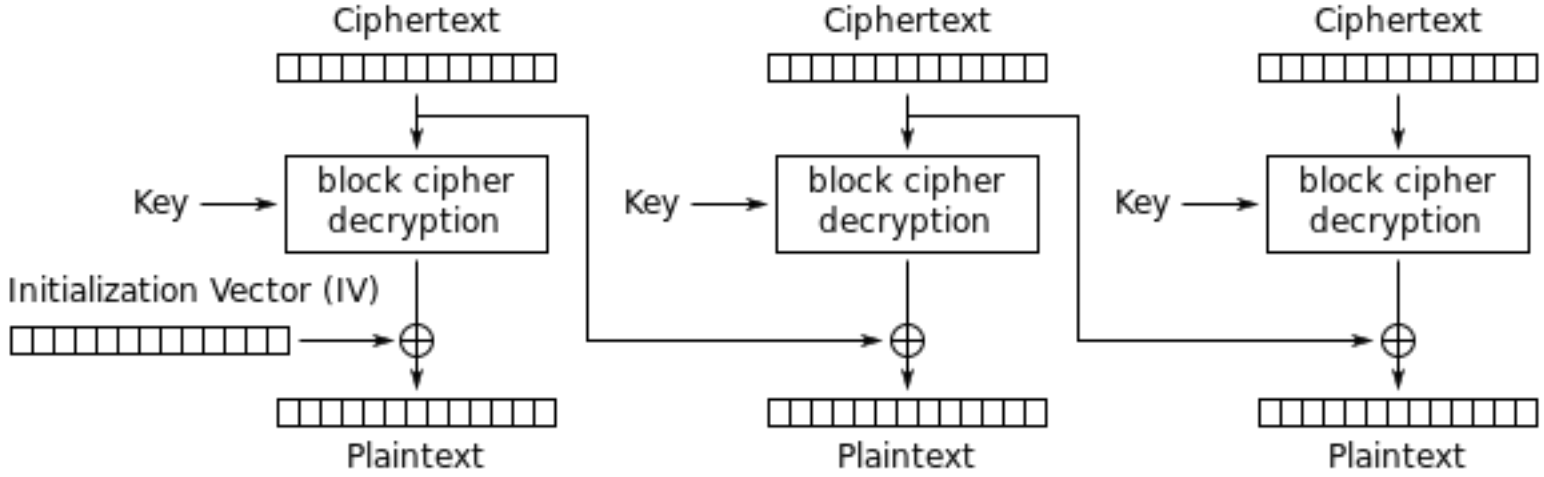
- US encryption standard [NIST 1993]
- 56-bit symmetric key, 64-bit plaintext input
- block cipher with cipher block chaining
- how secure is DES?
  - DES Challenge: 56-bit-key-encrypted phrase decrypted (brute force) in less than a day
  - no known good analytic attack
- making DES more secure:
  - 3DES: encrypt 3 times with 3 different keys

AES: Advanced Encryption Standard

- symmetric-key NIST standard, replaced DES (Nov 2001)
- processes data in 128 bit blocks
- 128, 192, or 256 bit keys
- brute force decryption (try each key) taking 1 sec on DES, takes 149 trillion years for AES



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Symmetric Cipher in Java

- Cipher Preparation
- Key and IV Preparation
- Encryption
- Decryption
- Random Number Generation

Block Cipher Padding

- Encryption algorithms typically handle data in blocks
  - DES uses a block size of 8 bytes
  - AES uses a block size of 16 bytes
- Data that does not have a length that is a multiple of block size must be padded with extra bytes
- PKCS #7 (Public-Key Cryptography Standard) is a commonly-used padding mechanism

  Suppose block size is 8 bytes, the last 1<=n<=8 bytes have a value of n
  - Example: if 2 bytes of padding are used then the last 2 bytes will be 2, 2; if no padding is needed, an extra block of 8, 8, 8, 8, 8, 8, 8, 8 will be needed.

Cipher Preparation

- Cipher name
  - Used to specify the characteristics of the desired cryptographic operation
  - Format: “algorithm/mode/padding”
  - Examples: “AES/CBC/PKCS7Padding”; “DES/ECB/NoPadding”
- Cipher objects
  - Obtained from a factory by specifying a cipher name and optionally a provider
  - Examples:

```
Cipher cipher = Cipher.getInstance(“AES/CBC/PKCS7Padding”, “BC”);
Cipher cipher = Cipher.getInstance(“AES/CBC/PKCS7Padding”);
```

Key and IV Preparation

- A pre-defined symmetric key can be encapsulated in a SecretKeySpec object

```
byte[] keyInBytes = new byte[] {.., .., …, ..}; //16 bytes
Key key128AES = new SecretKeySpec(keyInBytes, "AES");
```

- For modes that require an initialization vector (IV), it must be generated, stored in an IvParameterSpec object, and later used for both encryption and decryption

```
byte[] ivInBytes = new byte[]{.., …, ..}; //16 bytes
IvParameterSpec iv = new IvParameterSpec(ivInBytes);
```

Encryption

- A Cipher object is initialized with a key and ENCRYPT_MODE

```
cipher.init(Cipher.ENCRYPT_MODE, key128AES);
or
Cipher.init(Cipher.ENCRYPT_MODE,key128AES,iv);
```

- Encryption can proceed as one or more update() calls followed by a doFinal() call

```
int nCipherLen = cipher.update(
        plainTextInBytes, 0, plainTextInBytes.length,
        cipherTextInBytes, 0);
nCipherLen += cipher.doFinal(cipherTextInBytes, nCipherLen);
```

Decryption

- A Cipher object is initialized with a key and DECRYPT_MODE

```
cipher.init(Cipher.DECRYPT_MODE, key128AES);
or
Cipher.init(Cipher.ENCRYPT_MODE,key128AES,iv);
```

- Encryption can proceed as one or more update() calls followed by a doFinal() call

```
int nDecryptedTextLen = cipher.update(
        cipherTextInBytes, 0, cipherTextInBytes.length,
        decryptedTextInBytes, 0);
nDecryptedTextLen += cipher.doFinal(
        decryptedTextInBytes,
        nDecryptedTextLen);
```

Random Number Generation

- Key and IV should be generated using a cryptographically strong number generator using unpredictable seeds.
- In Java, the SecureRandom class is typically used to generate pseudo-random numbers; each SecureRandom object uses a new seed and should be re-instantiated when convenient
- Using SecureRandom to generate an IV

```
SecureRandom sr = new SecureRandom();
byte[] ivInBytes = new byte[16];
sr.nextBytes(ivInBytes);
```

- Random keys can be generated with a KeyGenerator object

```
KeyGenerator kg=keyGenerator.getInstance("AES");
kg.init(256);
Key key256AES = kg.generateKey();
```

Public Key Cryptography

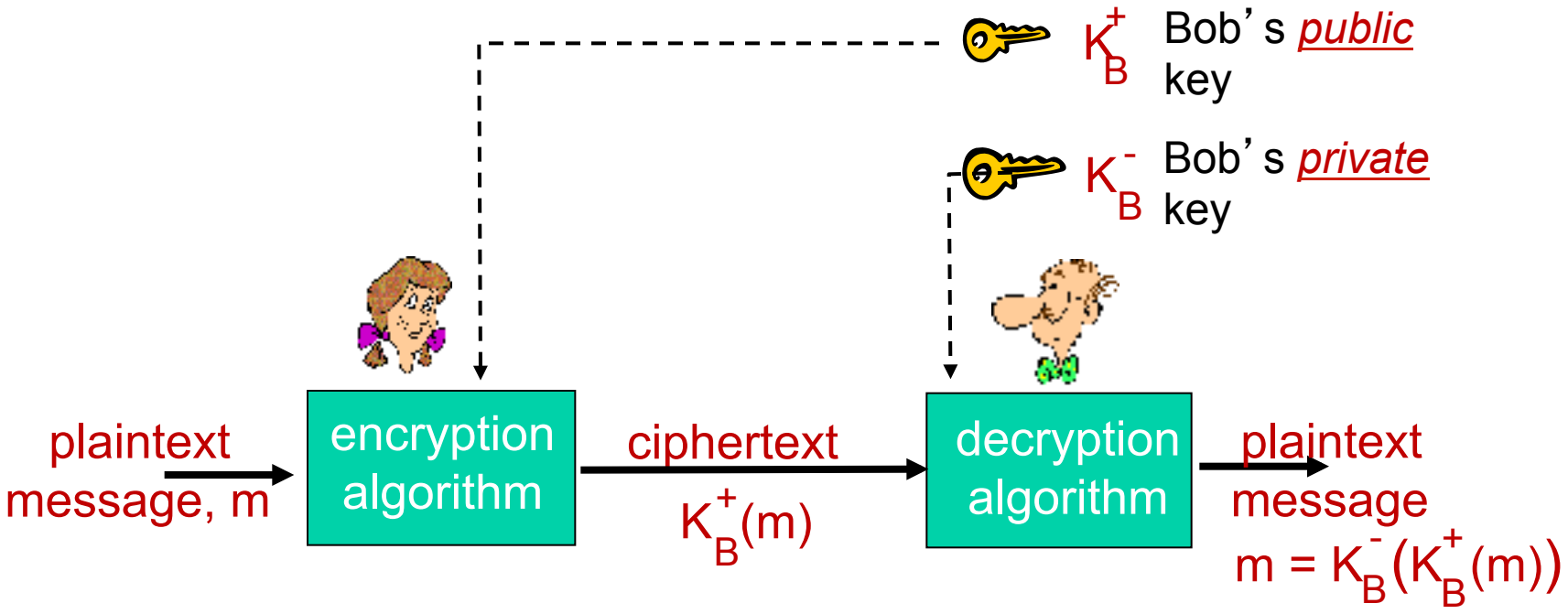
symmetric key crypto

- requires sender, receiver know shared secret key
- Q: how to agree on key in first place (particularly if never "met")?

public key crypto

- radically different approach [Diffie-Hellman76, RSA78]
- sender, receiver do *not* share secret key
- *public* encryption key known to *all*
- *private* decryption key known only to receiver

Public key cryptography



Public key encryption algorithms

requirements:

1. need $K_B^+(\cdot)$ and $K_B^-(\cdot)$ such that

$$K_B^-(K_B^+(m)) = m$$

2. given public key K_B^+, it should be impossible to compute private key K_B^-

RSA: Rivest, Shamir, Adelson algorithm

RSA: getting ready

- message: just a bit pattern
- bit pattern can be uniquely represented by an integer number
- thus, encrypting a message is equivalent to encrypting a number

example:

- m= 10010001 . This message is uniquely represented by the decimal number 145.
- to encrypt m, we encrypt the corresponding number, which gives a new number (the ciphertext).

RSA: Creating public/private key pair

1. choose two large prime numbers p, q. (e.g., 1024 bits each)
2. compute $n = pq$, $z = (p-1)(q-1)$
3. choose e (with $e<n$) that has no common factors with z (e, z are "relatively prime").
4. choose d such that $ed-1$ is exactly divisible by z. (in other words: ed mod $z = 1$).
5. *public* key is (n,e). *private* key is (n,d).

K_B^+ K_B^-

RSA: encryption, decryption

0. given (*n*,*e*) and (*n*,*d*) as computed above

1. to encrypt message *m* (<*n*), compute

$c = m^e \bmod n$

2. to decrypt received bit pattern, *c*, compute

$m = c^d \bmod n$

magic happens!

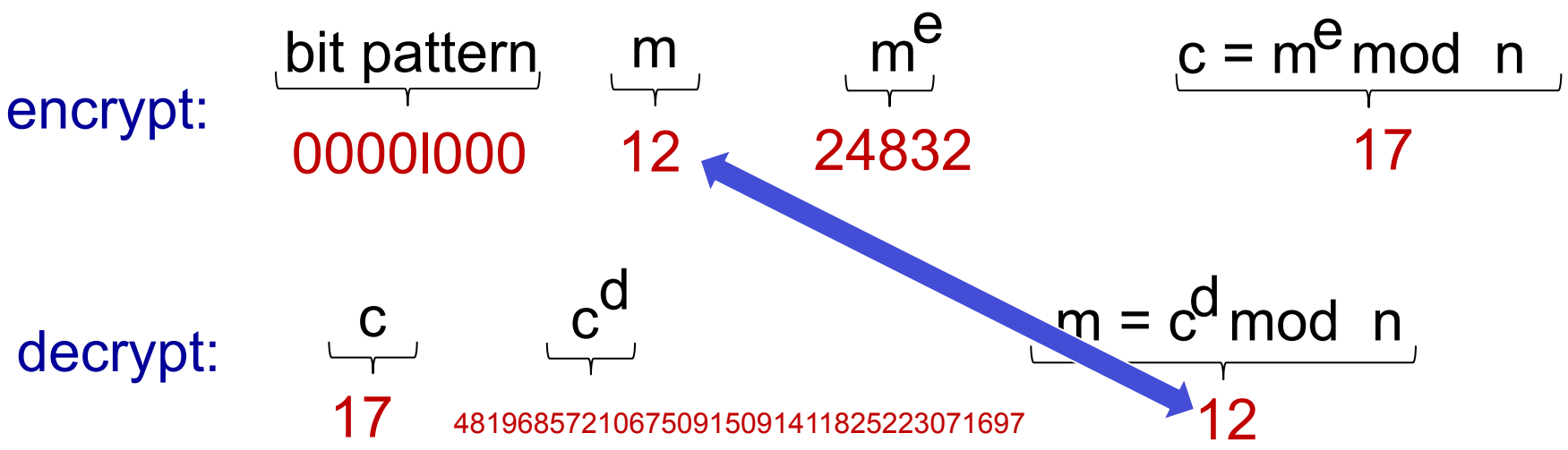
$$m = \underbrace{(m^e \bmod n)}_{c}{}^d \bmod n$$

RSA example:

Bob chooses *p=5, q=7*. Then *n=35, z=24*.

e=5 (so *e, z* relatively prime).
d=29 (so *ed-1* exactly divisible by *z*).

encrypting 8-bit messages.



RSA: another important property

The following property will be *very* useful later:

$$\underbrace{K_B^-(K_B^+(m))}_{\text{use public key first, followed by private key}} = m = \underbrace{K_B^+(K_B^-(m))}_{\text{use private key first, followed by public key}}$$

result is the same!

Why is RSA secure?

- suppose you know Bob's public key (n,e). How hard is it to determine d?
- essentially need to find factors of n without knowing the two factors p and q
  - fact: factoring a big number is hard

RSA in practice: session keys

- exponentiation in RSA is computationally intensive
- DES is at least 100 times faster than RSA
- use public key crypto to establish secure connection, then establish second key – symmetric session key – for encrypting data

session key, K_S

- Bob and Alice use RSA to exchange a symmetric key K_S
- once both have K_S, they use symmetric key cryptography

Key-pair Generation

```
SecureRandom rand = new SecureRandom( );

KeyPairGenerator kpg = KeyPairGenerator.getInstance(
        "RSA");
kpg.initialize(2048, rand); //specify the length of key
KeyPair kp = kpg.generateKeyPair( );

PublicKey keyPublic = kp.getPublic( );
PrivateKey keyPrivate = kp.getPrivate( );
```

Encryption with Public Key

```
Cipher cipherEncrypt = Cipher.getInstance( "RSA/NONE/
OAEPPading" ); //OAEPPading is typically used with RSA

cipherEncrypt.init(Cipher.ENCRYPT_MODE, keyPublic,
rand);

byte[] bytesEncrypted = cipherEncrypt.doFinal(bytesOriginal);
```

Decryption with Public Key

```
Cipher cipherDecrypt = Cipher.getInstance( "RSA/NONE/
OAEPPading" ); //OAEPPading is typically used with RSA

cipherDecrypt.init(Cipher.DECRYPT_MODE, keyPrivate);

byte[] bytesDecrypted =
cipherDecrypt.doFinal(bytesEncrypted);
```

Network Security I

1. What is network security?
2. Principles of cryptography
3. Message integrity, *authentication*
4. Secure socket programming

Authentication

Goal: Bob wants Alice to "prove" her identity to him

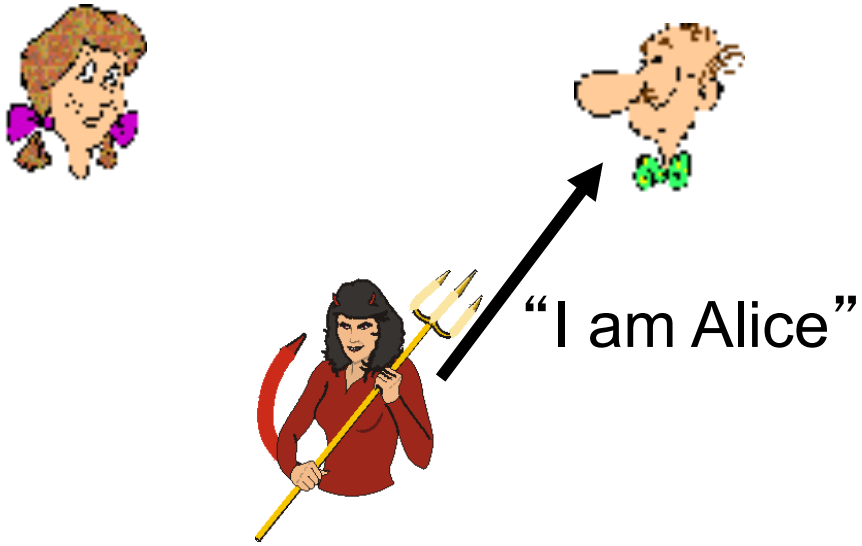
Protocol ap1.0: Alice says "I am Alice"

Failure scenario??

Authentication

Goal: Bob wants Alice to "prove" her identity to him

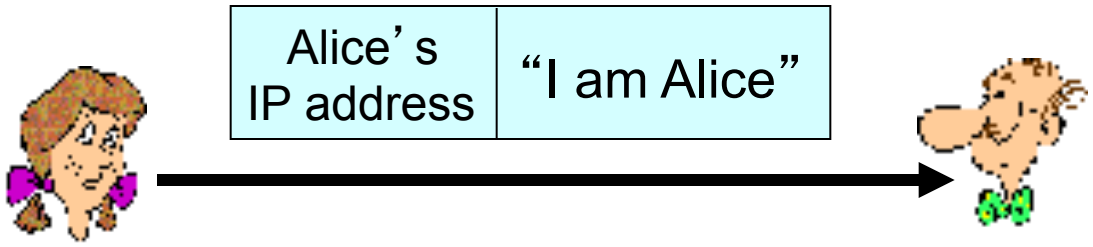
Protocol ap1.0: Alice says "I am Alice"



in a network,
Bob can not "see" Alice,
so Trudy simply declares
herself to be Alice

Authentication: another try

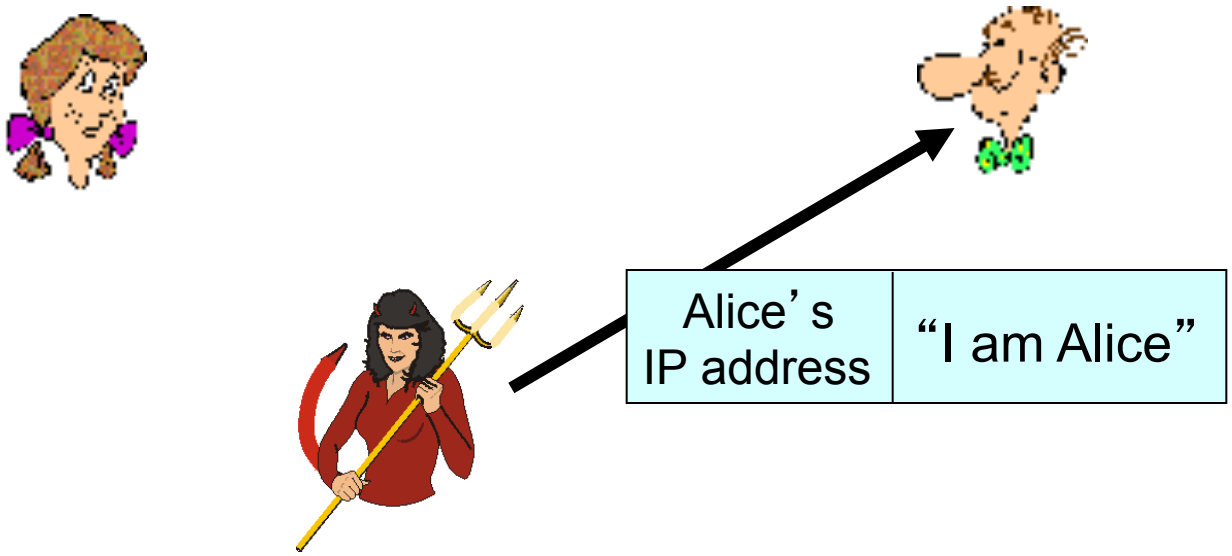
Protocol ap2.0: Alice says "I am Alice" in an IP packet containing her source IP address



Failure scenario??

Authentication: another try

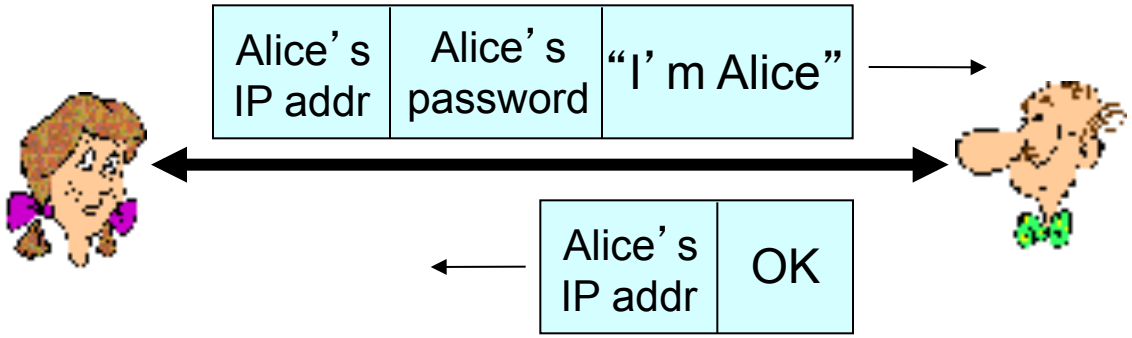
Protocol ap2.0: Alice says "I am Alice" in an IP packet containing her source IP address



Trudy can create a packet "spoofing" Alice's address

Authentication: another try

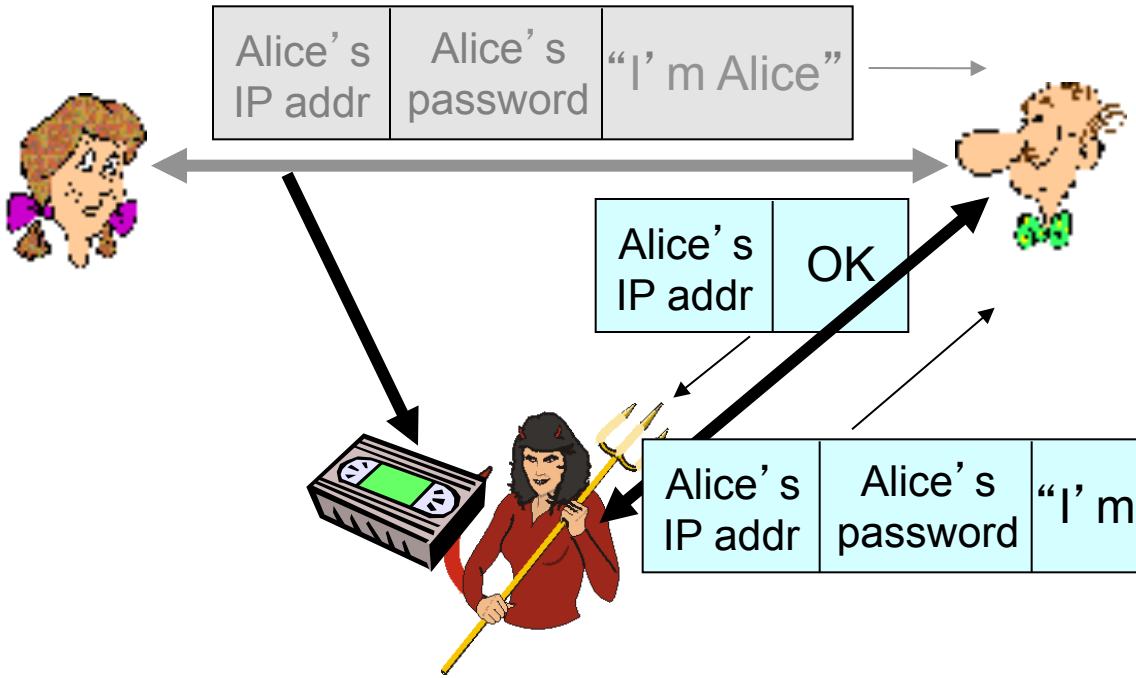
Protocol ap3.0: Alice says "I am Alice" and sends her secret password to "prove" it.



Failure scenario??

Authentication: another try

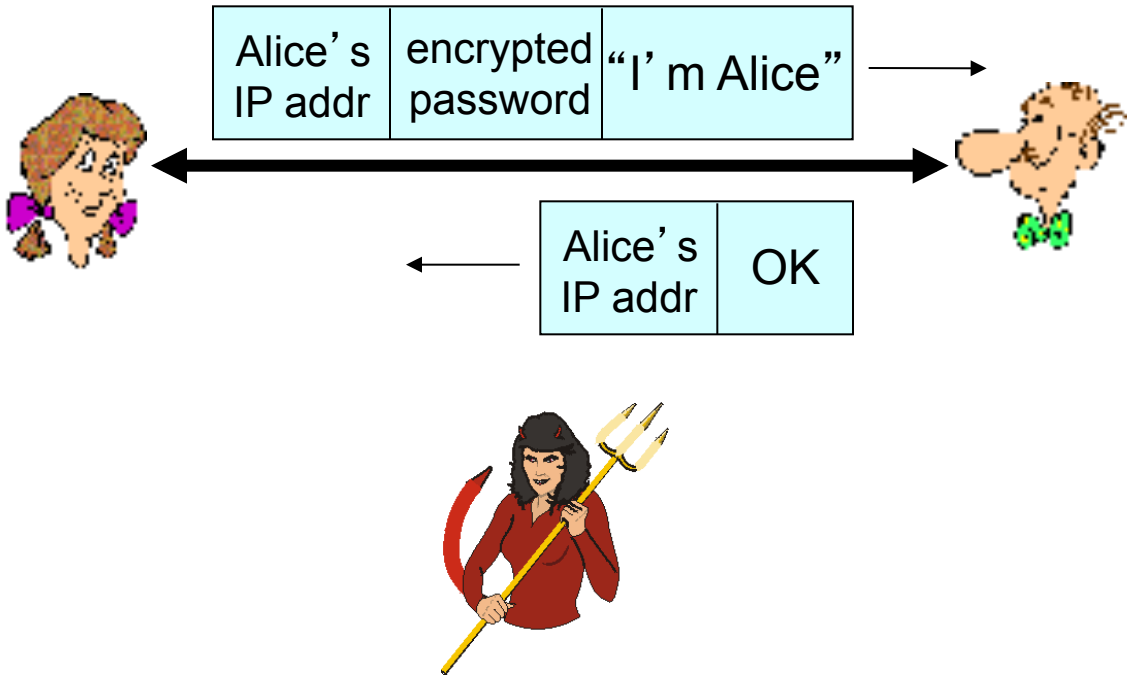
Protocol ap3.0: Alice says "I am Alice" and sends her secret password to "prove" it.



playback attack: Trudy records Alice's packet and later plays it back to Bob

Authentication: yet another try

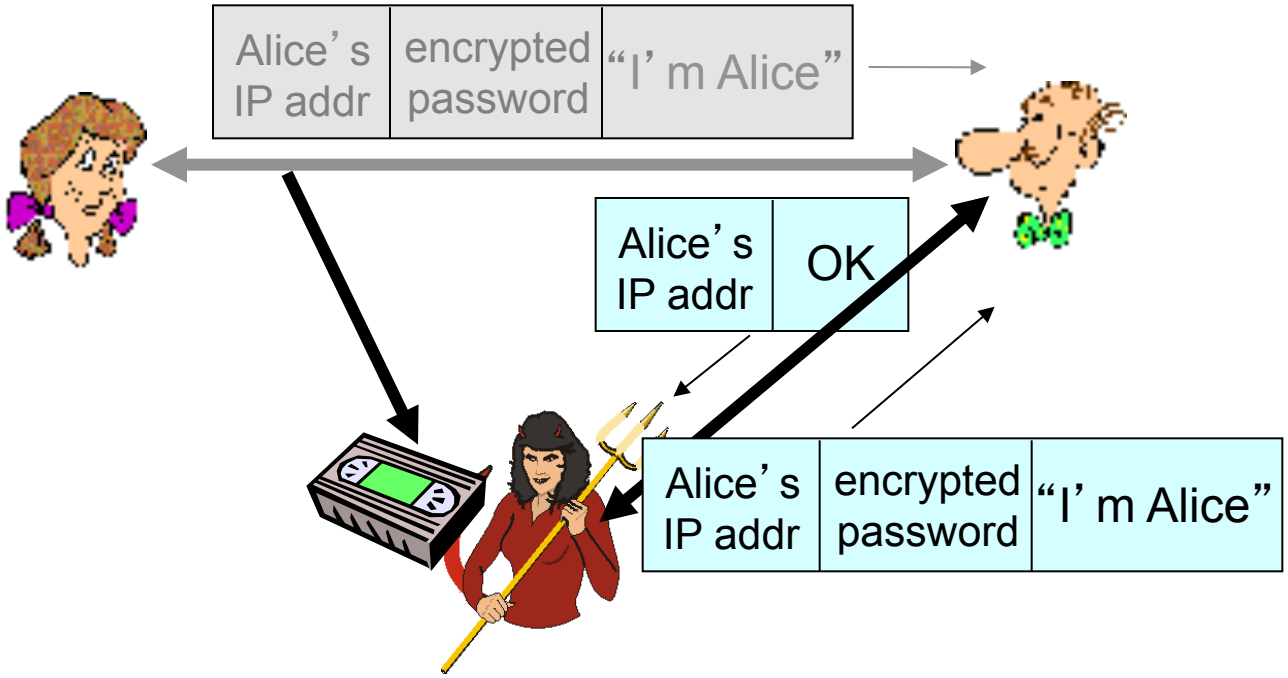
Protocol ap3.1: Alice says "I am Alice" and sends her *encrypted* secret password to "prove" it.



Failure scenario??

Authentication: yet another try

Protocol ap3.1: Alice says "I am Alice" and sends her *encrypted* secret password to "prove" it.



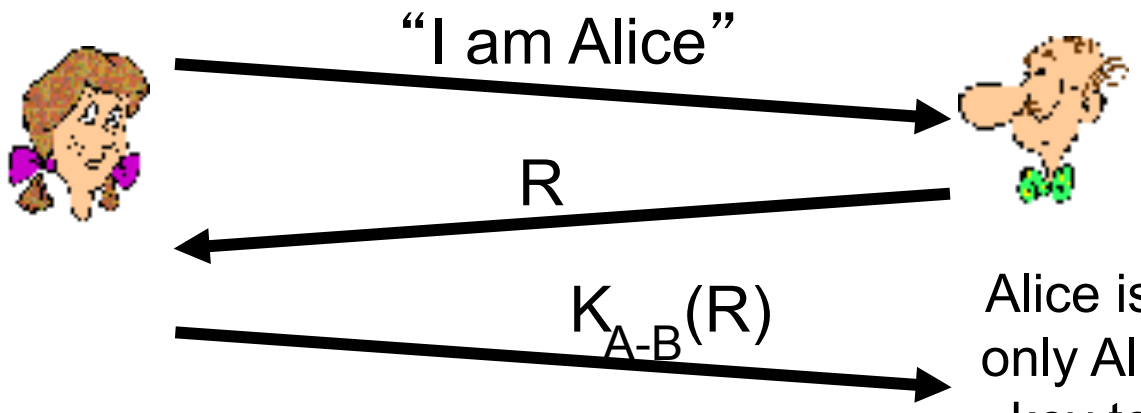
record and playback *still* works!

Authentication: yet another try

Goal: avoid playback attack

nonce: number (R) used only *once-in-a-lifetime*

ap4.0: to prove Alice "live", Bob sends Alice *nonce*, R. Alice must return R, encrypted with shared secret key



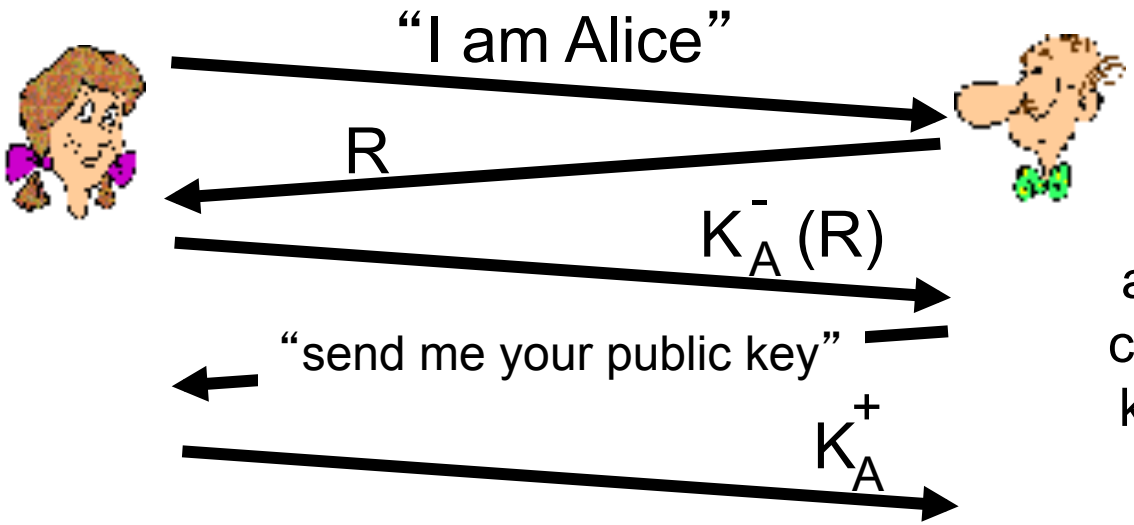
Failures, drawbacks?

Authentication: ap5.0

ap4.0 requires shared symmetric key

- can we authenticate using public key techniques?

ap5.0: use nonce, public key cryptography



Bob computes

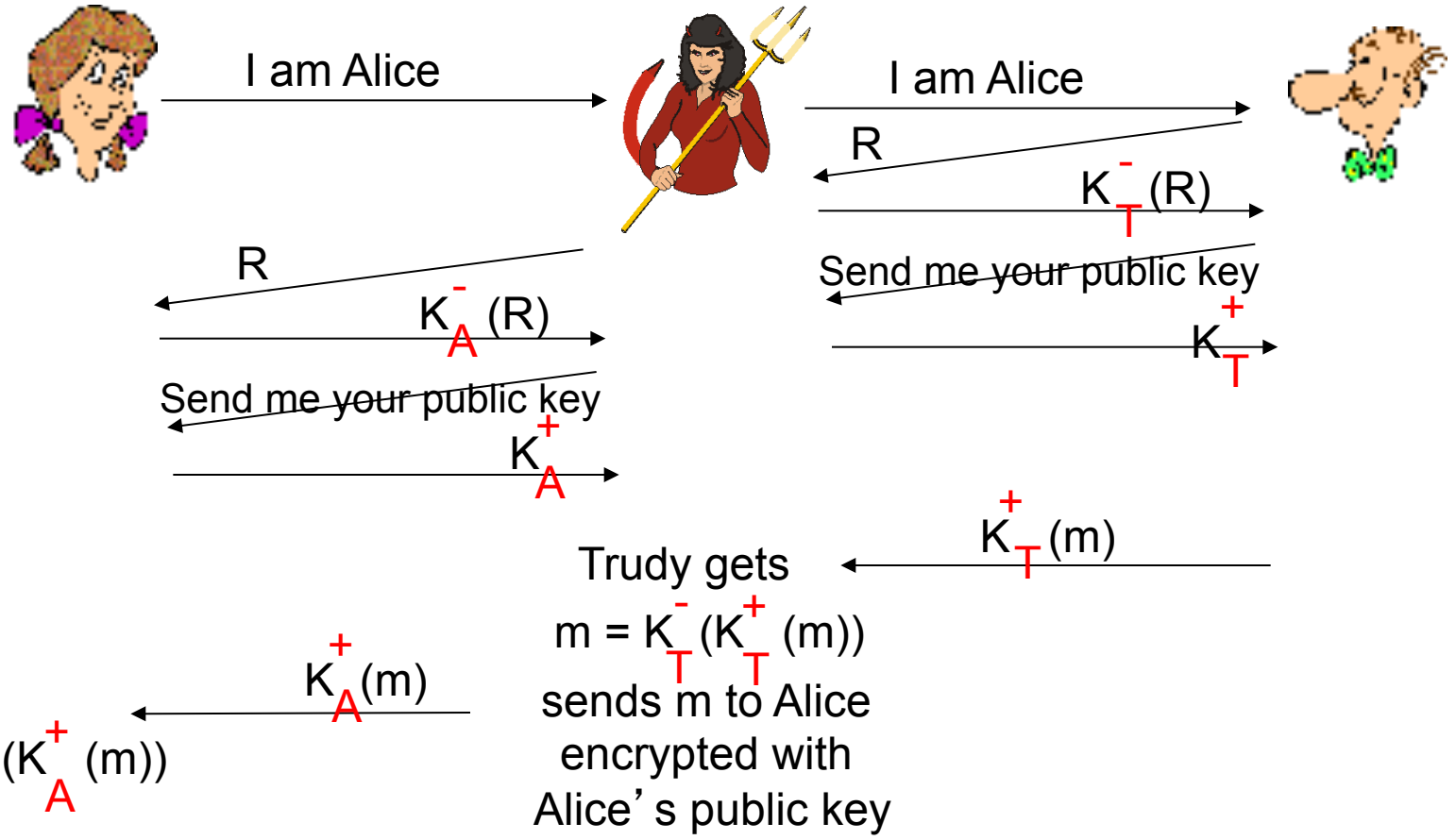
$K_A^+(K_A^-(R)) = R$

and knows only Alice could have the private key, that encrypted R such that

$K_A^+(K_A^-(R)) = R$

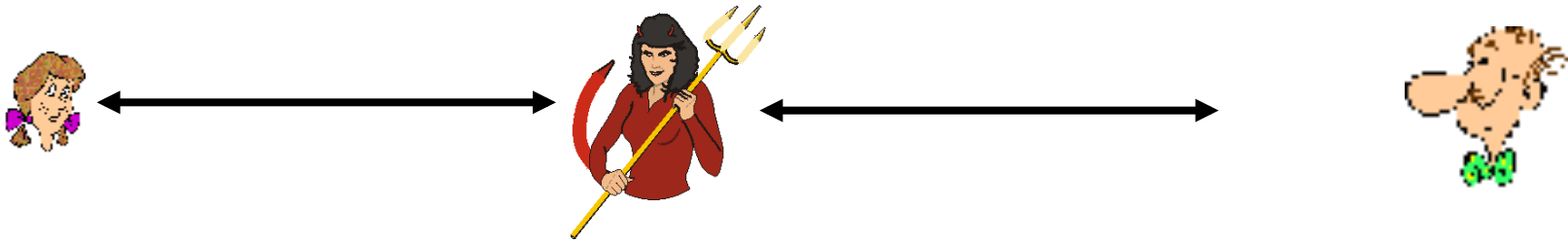
ap5.0: security hole

man (or woman) in the middle attack: Trudy poses as Alice (to Bob) and as Bob (to Alice)



ap5.0: security hole

man (or woman) in the middle attack: Trudy poses as Alice (to Bob) and as Bob (to Alice)



difficult to detect:

- Bob receives everything that Alice sends, and vice versa. (e.g., so Bob, Alice can meet one week later and recall conversation!)
- problem is that Trudy receives all messages as well!

Network Security I

1. What is network security?
2. Principles of cryptography
3. *Message integrity,* authentication
4. Securing e-mail

Digital signatures

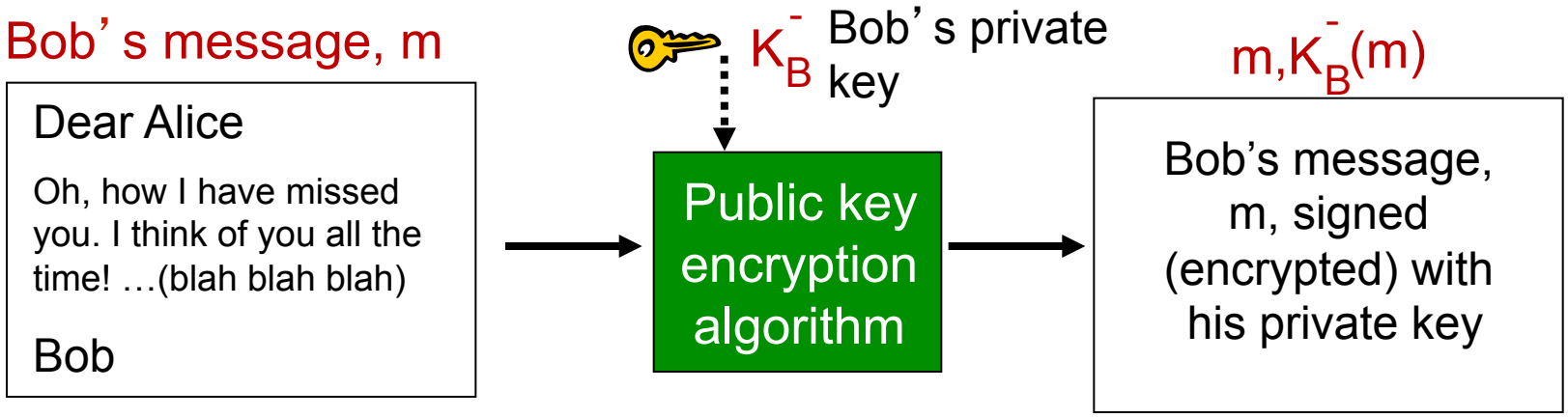
cryptographic technique analogous to hand-written signatures:

- sender (Bob) digitally signs document, establishing he is document owner/creator.
- *verifiable, nonforgeable:* recipient (Alice) can prove to someone that Bob, and no one else (including Alice), must have signed document

Digital signatures

simple digital signature for message m:

- Bob signs m by encrypting with his private key K_B^-, creating "signed" message, $K_B^-(m)$



Digital signatures

- suppose Alice receives msg m, with signature: m, $K_B^-(m)$
- Alice verifies m signed by Bob by applying Bob's public key K_B^+ to $K_B^-(m)$ then checks $K_B^+(K_B^-(m)) = m$.
- If $K_B^+(K_B^-(m)) = m$, whoever signed m must have used Bob's private key.

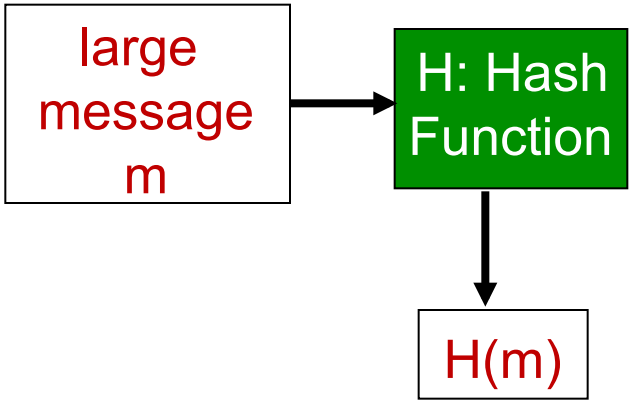
Alice thus verifies that:

- Bob signed m
- no one else signed m
- Bob signed m and not m '

non-repudiation:

- ✓ Alice can take m, and signature $K_B^-(m)$ to court and prove that Bob signed m

Message digests



computationally expensive to public-key-encrypt long messages

goal: fixed-length, easy-to-compute digital "fingerprint"

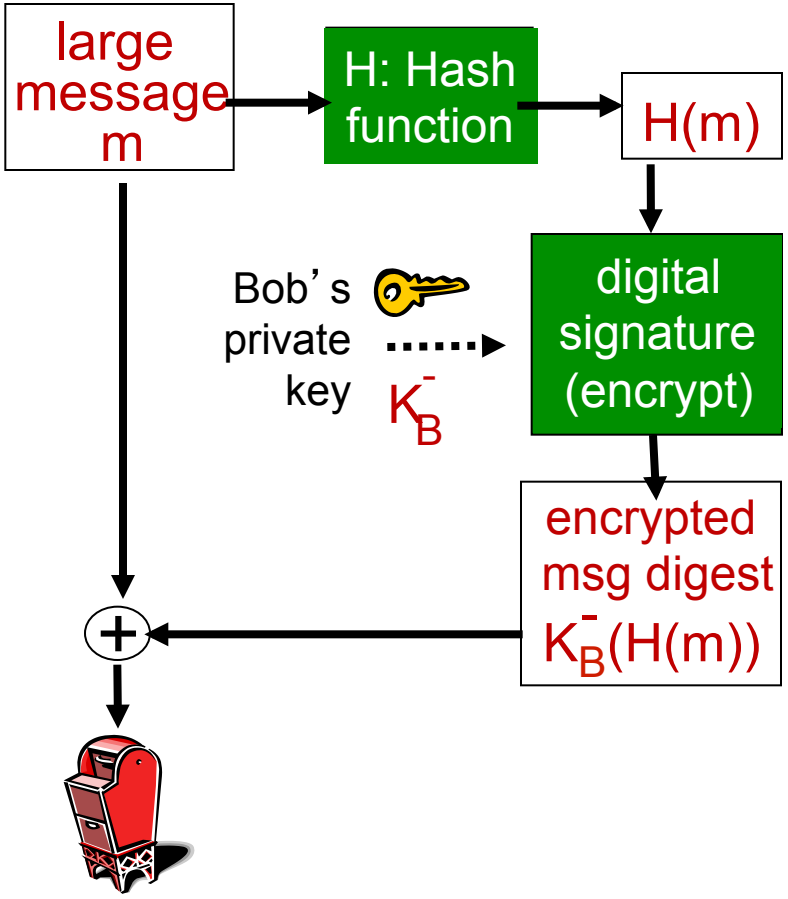
- apply hash function H to *m*, get fixed size message digest, *H(m)*.

Hash function properties:

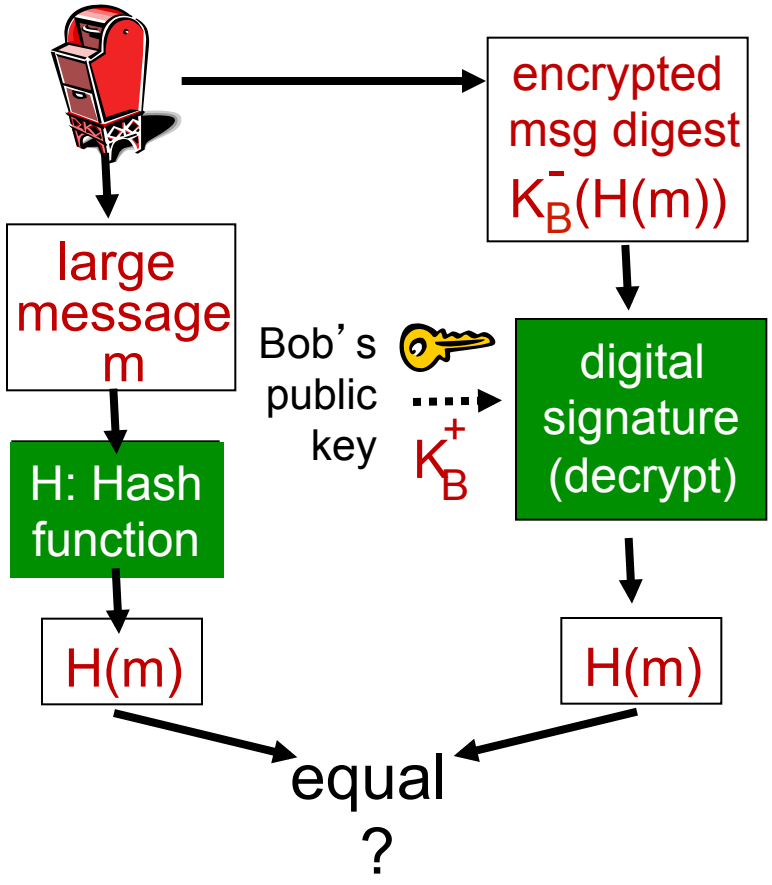
- many-to-1
- produces fixed-size msg digest (fingerprint)
- given message digest x, computationally infeasible to find m such that x = H(m)

Digital signature = signed message digest

Bob sends digitally signed message:



Alice verifies signature, integrity of digitally signed message:



Hash function algorithms

- MD5 hash function widely used (RFC 1321)
  - computes 128-bit message digest in 4-step process.
  - arbitrary 128-bit string x, appears difficult to construct msg m whose MD5 hash is equal to x
- SHA-1 is also used
  - US standard [NIST, FIPS PUB 180-1]
  - 160-bit message digest
- SHA-224, SHA-256, SHA-384, SHA-512
  - Designed for compatibility with increased security provided by the AES cipher
- NIST issued FIPS 202 to announce SHA-3 Standard in 2015
  - Permutation-based Hash and Extendable Output Functions

Java Programming with Hash Functions

- Java provides class java.security.MessageDigest, which encapsulates computing a message digest

- To instantiate an object of MessageDigest class:
  - MessageDigest.getInstance("SHA-256");
- To pass in the message to digest
  - MessageDigest.update(String);
- To return the digest
  - MessageDigest.digest();

- To sign a message, we need the private key of the sender and the signature specification, e.g., SHA512withRSA:

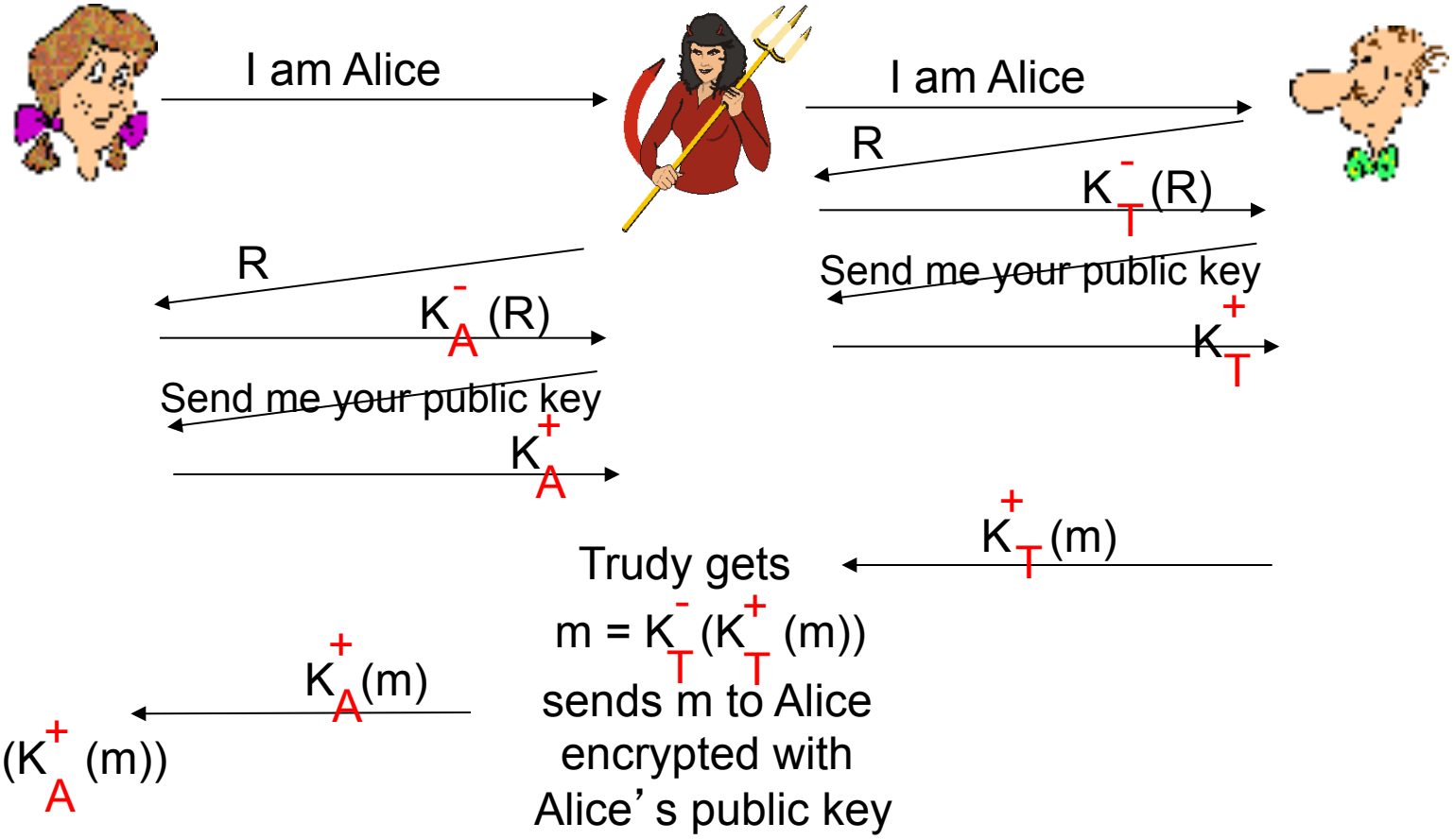
```
byte[] bytesMessage = strMessage.getBytes("UTF-8");
Signature sigSender = Signature.getInstance("SHA512withRSA");
sigSender.initSign(privateKey);
sigSender.update(bytesMessage);
byte[] bytesSignature = sigSender.sign();
```

- To verify a digital signature, we need the public key of the sender:

```
Signature sigReceiver = Signature.getInstance("SHA512withRSA");
sigReceiver.initVerify(publicKey);
sigReceiver.update(byteMessage);
boolean bValid = sigReceiver.verify(byteSignature);
```

Recall: ap5.0 security hole

man (or woman) in the middle attack: Trudy poses as Alice (to Bob) and as Bob (to Alice)

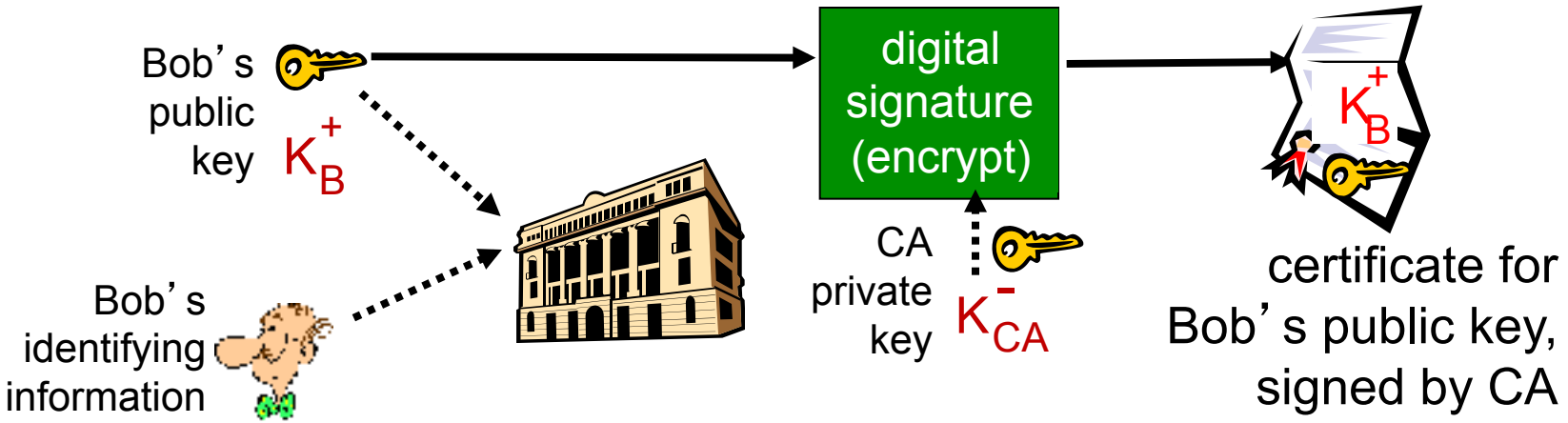


Public-key certification

- motivation: Trudy plays pizza prank on Bob
  - Trudy creates e-mail order:
    Dear Pizza Store, Please deliver to me four pepperoni pizzas. Thank you, Bob
  - Trudy signs order with her private key
  - Trudy sends order to Pizza Store
  - Trudy sends to Pizza Store her public key, but says it's Bob's public key
  - Pizza Store verifies signature; then delivers four pepperoni pizzas to Bob
  - Bob doesn't even like pepperoni

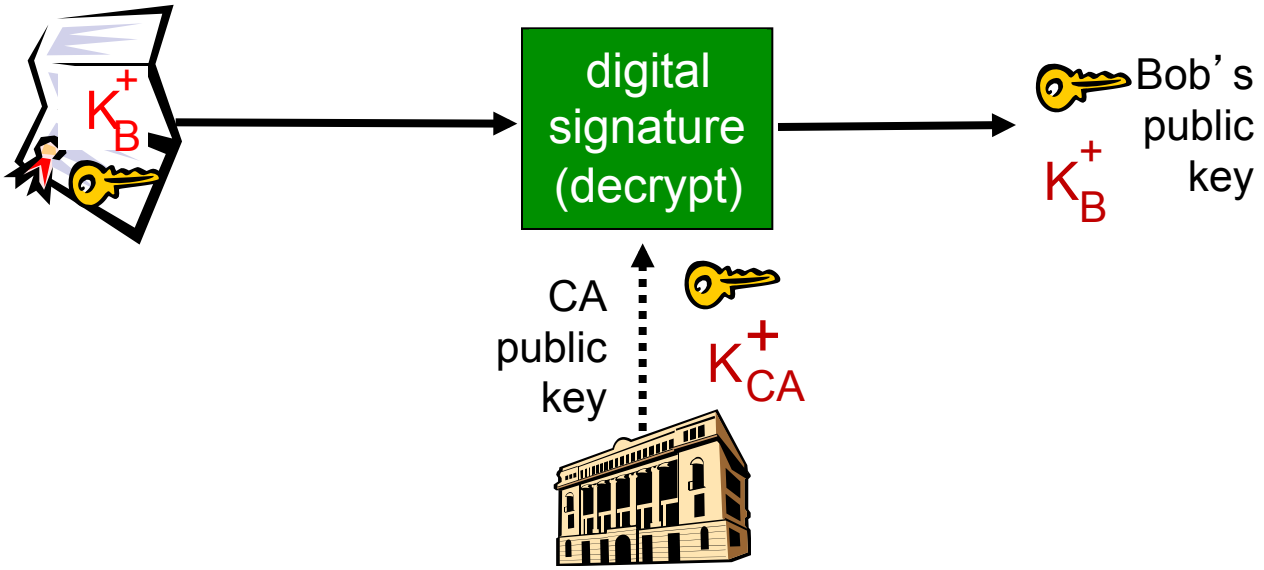
Certification authorities

- *certification authority (CA):* binds public key to particular entity, E.
- E (person, router) registers its public key with CA.
  - E provides "proof of identity" to CA.
  - CA creates certificate binding E to its public key.
  - certificate containing E's public key digitally signed by CA – CA says "this is E's public key"



Certification authorities

- when Alice wants Bob's public key:
  - gets Bob's certificate (Bob or elsewhere).
  - apply CA's public key to Bob's certificate, get Bob's public key



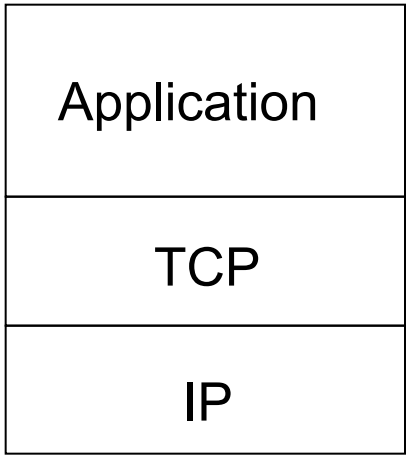
Network Security I

1. What is network security?
2. Principles of cryptography
3. Message integrity
4. *Secure socket programming*

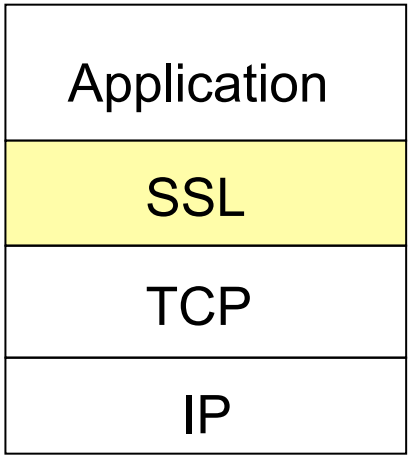
SSL: Secure Sockets Layer

- widely deployed security protocol
  - supported by almost all browsers, web servers
  - https
  - billions $/year over SSL
- variation -TLS: transport layer security, RFC 2246
- provides
  - *confidentiality*
  - *integrity*
  - *authentication*
- original goals:
  - Web e-commerce transactions
  - encryption (especially credit-card numbers)
  - Web-server authentication
  - optional client authentication
  - minimum hassle in doing business with new merchant
- available to all TCP applications
  - secure socket interface

SSL and TCP/IP



normal application



application with SSL

- SSL provides application programming interface (API) to applications
- C and Java SSL libraries/classes readily available

SSL cipher suite

- cipher suite
  - public-key algorithm
  - symmetric encryption algorithm
  - MAC algorithm
- SSL supports several cipher suites
- negotiation: client, server agree on cipher suite
  - client offers choice
  - server picks one

common SSL symmetric ciphers

- DES – Data Encryption Standard: block
- 3DES – Triple strength: block
- RC2 – Rivest Cipher 2: block
- RC4 – Rivest Cipher 4: stream

SSL Public key encryption

- RSA

SSL: handshake (1)

Purpose

1. server authentication
2. negotiation: agree on crypto algorithms
3. establish keys
4. client authentication (optional)

SSL: handshake (2)

1. client sends list of algorithms it supports, along with client nonce
2. server chooses algorithms from list; sends back: choice + certificate + server nonce
3. client verifies certificate, extracts server's public key, generates pre_master_secret, encrypts with server's public key, sends to server
4. client and server independently compute encryption and MAC keys from pre_master_secret and nonces
5. client sends a MAC of all the handshake messages
6. server sends a MAC of all the handshake messages

SSL: handshaking (3)

last 2 steps protect handshake from tampering

- client typically offers range of algorithms, some strong, some weak
- man-in-the middle could delete stronger algorithms from list
- last 2 steps prevent this
  - last two messages are encrypted

SSL: handshaking (4)

- why two random nonces?
- suppose Trudy sniffs all messages between Alice & Bob
- next day, Trudy sets up TCP connection with Bob, sends exact same sequence of records
  - Bob (Amazon) thinks Alice made two separate orders for the same thing
  - solution: Bob sends different random nonce for each connection. This causes encryption keys to be different on the two days
  - Trudy's messages will fail Bob's integrity check

Key derivation

- client nonce, server nonce, and pre-master secret input into pseudo random-number generator.
  - produces master secret
- master secret and new nonces input into another random-number generator: "key block"
  - because of resumption: TBD
- key block sliced and diced:
  - client MAC key
  - server MAC key
  - client encryption key
  - server encryption key
  - client initialization vector (IV)
  - server initialization vector (IV)

Network Security (summary)

basic techniques…...

- cryptography (symmetric and public)
- message integrity
- end-point authentication

…. used in many different security scenarios

- secure email
- secure transport (SSL)
- IP sec
- 802.11

operational security: firewalls and IDS

SSL Client/Server Programming Example

Writing a simple SSL Client

- All SSL client must have a truststore:
  - ✓ holds trusted certificates (signed public keys of CA's)
  - ✓ in the same format as a keystore (a store for public/private key pair)
  - ✓ an instance of Java's KeyStore class
  - ✓ used by the client to verify the certificate sent by the server
  - ✓ may be shared with others

- If a client is to be verified by the server then the client needs a keystore as well as a trustore

Creating a Truststore

- Use keytool –genkey to create an RSA key pair
- Use keytool –export to generate a self-signed RSA certificate (holding no private key)
- Use keytool –import to place the certificate into a truststore

Use keytool –genkey to create an RSA key pair

~\keystoreexamples> keytool -genkey -alias wz -keyalg RSA -keystore wzkeystore

Enter keystore password: password

What is your first and last name?
[Unknown]: Wensheng Zhang

What is the name of your organizational unit?
[Unknown]: Computer Science Department

What is the name of your organization?
[Unknown]: Iowa State University

What is the name of your City or Locality?
[Unknown]: Ames

What is the name of your State or Province?
[Unknown]: IA

What is the two-letter country code for this unit?
[Unknown]: US

Is CN=Wensheng Zhang, OU=Computer Science Department, O=Iowa State University, L=Ames, ST=IA, C=US correct?
[no]: yes

Enter key password for <wz>
(RETURN if same as keystore password): <RT>

Use keytool –export to generate a self-signed RSA certificate (holding no private key)

~\keystoreexamples>keytool -export -alias wz -keystore wzkeystore -file wz.cert
Enter keystore password: password
Certificate stored in file <wz.cert>

Use keytool –import to place the certificate into a truststore

~\keystoreexamples>keytool -import -alias wz -keystore wz.truststore -file wz.cert

Enter keystore password: password
Owner:
CN=Wensheng Zhang, OU=Computer Science Department, O=Iowa State University, L=Ames, ST=IA, C=US

Issuer:
CN=Wensheng Zhang, OU=Computer Science Department, O=Iowa State University, L=Ames, ST=IA, C=US

Serial number: 3a6aee2c
Valid from: Sun Jan 20 14:23:27 CST 2019 until: Sat Apr 20 15:23:27 CDT 2019
Certificate fingerprints:
MD5: 1E:BF:E9:7F:FC:F8:6C:72:7F:D1:E9:FA:04:6E:86:A9
SHA1: 02:82:5D:E9:D4:9C:EA:DE:
86:FE:D7:14:00:85:FF:2C:A7:16:E5:11
SHA256: AF:B8:8B:AD:EA:EF:D3:FB:04:8D:
6C:F7:66:E4:82:BB:BF:A3:CF:E0:0B:1C:91:AE:B1:0C:CB:
73:A1:E6:A7:C8
Signature algorithm name: SHA256withRSA
Version: 3

Trust this certificate? [no]: yes
Certificate was added to keystore

wzkeystore will be placed in the server's directory
SSL will send the associated certificate to the client

wz.truststore will be placed in the client's directory

Client.java

```
import java.io.*;
import javax.net.ssl.*;
import java.net.*;
import javax.net.*;

public class Client {

    public static void main(String args[]) {

        int port = 6789;
        try {
            // tell the system who we trust
            System.setProperty("javax.net.ssl.trustStore", "wz.truststore");
```

```
// get an SSLSocketFactory
SocketFactory sf = SSLSocketFactory.getDefault();

// an SSLSocket "is a" Socket
Socket s = sf.createSocket("localhost", 6789);

PrintWriter out = new PrintWriter(s.getOutputStream());
BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                        s.getInputStream()));
out.write("Hello server\n");
out.flush();
String answer = in.readLine();
System.out.println(answer);
```

```
            out.close();
            in.close();
        }
        catch(Exception e) {
            System.out.println("Exception thrown " + e);
        }
    }
}
```

Server.java

```
// Server side SSL
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;
import java.security.*;

public class Server {

    // hold the name of the keystore containing public and private keys
    static String keyStore = "wzkeystore";

    // password of the keystore (same as the alias)
    static char keyStorePass[] = "password".toCharArray();
```

```
public static void main(String args[]) {

    int port = 6789;
    SSLServerSocket server;

    try {
        // get the keystore into memory
        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream(keyStore), keyStorePass);

        // initialize the key manager factory with the keystore data
        KeyManagerFactory kmf =
                            KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks,keyStorePass);
```

```
// initialize the SSLContext engine
// may throw NoSuchProvider or NoSuchAlgorithm exception
// TLS - Transport Layer Security most generic

SSLContext sslContext = SSLContext.getInstance("TLS");

// Inititialize context with given KeyManagers, TrustManagers,
// SecureRandom defaults taken if null

sslContext.init(kmf.getKeyManagers(), null, null);

// Get ServerSocketFactory from the context object
ServerSocketFactory ssf = sslContext.getServerSocketFactory();
```

```
//  Now like programming with normal server sockets
ServerSocket serverSocket = ssf.createServerSocket(port);

System.out.println("Accepting secure connections");

Socket client = serverSocket.accept();
System.out.println("Got connection");

BufferedWriter out = new BufferedWriter(
                                 new OutputStreamWriter(
                                        client.getOutputStream()));
BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                        client.getInputStream()));
```

```
            String msg = in.readLine();
            System.out.println("Got message " + msg);
            out.write("Hello client\n");
            out.flush();
            in.close();
            out.close();


        }
        catch(Exception e) {
            System.out.println("Exception thrown " + e);
        }
    }
}
```

On the server

```
~\...\server>java Server
Accepting secure connections
Got connection
Got message Hello server
```

On the client

```
~\...\client>java Client
Hello client
```

What we have so far…

The Client

- Has a list of public keys it trusts in the file wz.truststore
- Has no public/private key pair of its own

The Server

- Has no list of trusted public keys in a truststore
- Has a public/private key pair of its own

For client authentication we need

- To generate a key pair for the client
- Extract a client certificate from the key pair
- Import this certificate into the server's truststore
- Have the server code trust the truststore
- Have the client code know about its own keys

Generate a key pair for the client

keytool -genkey -alias wzclient -keyalg RSA -keystore wzclientkeystore

Extract a client certificate from the key pair

keytool -export -alias wzclient -keystore wzclientkeystore -file wzclient.cert

Import the certificate into the server's truststore

keytool -import -alias wzclient -keystore wzclient.truststore
-file wzclient.cert

Have the server code trust the truststore

```
// Server side SSL
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;
import java.security.*;

public class Server {
    // hold the name of the keystore containing public and private keys
    static String keyStore = "wzkeystore";

    // password of the keystore (same as the alias)
    static char keyStorePass[] = ”password".toCharArray();
```

```
public static void main(String args[]) {

    int port = 6789;
    SSLServerSocket server;

    try {
        // get the keystore into memory
        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream(keyStore), keyStorePass);

        // initialize the key manager factory with the keystore data
        KeyManagerFactory kmf =
                KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks,keyStorePass);
```

```
// tell the system who we trust, we trust the client's certificate

System.setProperty("javax.net.ssl.trustStore", "wzclient.truststore");

// initialize the SSLContext engine

// may throw NoSuchProvider or NoSuchAlgorithm exception
// TLS - Transport Layer Security most generic

SSLContext sslContext = SSLContext.getInstance("TLS");
// Inititialize context with given KeyManagers, TrustManagers,
// SecureRandom
// defaults taken if null
sslContext.init(kmf.getKeyManagers(), null, null);
```

```
// Get ServerSocketFactory from the context object
   ServerSocketFactory ssf = sslContext.getServerSocketFactory();

//  Now almost like programming with normal server sockets
    ServerSocket serverSocket = ssf.createServerSocket(port);
    ((SSLServerSocket)serverSocket).setNeedClientAuth(true);
    System.out.println("Accepting secure connections");
    Socket client = serverSocket.accept();
    System.out.println("Got connection");
    PrintWriter out = new
                          PrintWriter(client.getOutputStream(),true);

    BufferedReader in = new BufferedReader(
          new InputStreamReader(client.getInputStream()));
```

```
            String fromClient = in.readLine();
            System.out.println(fromClient);
            out.println("Hello client\n");
            out.flush();
            in.close();
            out.close();
            System.out.println("Data sent");

        }
        catch(Exception e) {
            System.out.println("Exception thrown " + e);
        }
    }
}
```

Have the client code know about its own keys

```
import java.net.*;
import java.io.*;
import javax.net.ssl.*;
import javax.security.cert.X509Certificate;
import java.security.KeyStore;

public class Client {

    public static void main(String args[]) {
        int port = 6789;
        // tell the system who we trust
        System.setProperty("javax.net.ssl.trustStore",”wz.truststore");
```

```
try {
      SSLSocketFactory factory = null;
      try {
         SSLContext ctx;
         KeyManagerFactory kmf;
         KeyStore ks;
         char[] passphrase = "password".toCharArray();
         ctx = SSLContext.getInstance("TLS");
         kmf = KeyManagerFactory.getInstance("SunX509");
         ks = KeyStore.getInstance("JKS");
         ks.load(new FileInputStream(”wzclientkeystore"), passphrase);
         kmf.init(ks, passphrase);
         ctx.init(kmf.getKeyManagers(), null, null);
         factory = ctx.getSocketFactory();
      } catch (Exception e) {throw new IOException(e.getMessage());
```

```
SSLSocket s = (SSLSocket)factory.createSocket("localhost", port);
s.startHandshake();
PrintWriter out = new PrintWriter(s.getOutputStream());
BufferedReader in = new BufferedReader(new InputStreamReader(
                                        s.getInputStream()));
out.write("Hello server\n");
out.flush();
String answer = in.readLine();
System.out.println(answer);
out.close();
in.close();
}catch(Exception e){System.out.println("Exception thrown " + e); }
}
}
```

Testing

~\…\server>
java Server
Accepting secure connections
Got connection
Hello server
Data sent

~\…\client>java Client
Hello client

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR