COMP2221 Networks

David Head

University of Leeds

Lecture 13

Previous lectures

In the last few lectures we have seen how a server can be implemented to handle multiple clients simultaneously:

- **Multi-threaded** servers in which each client handler runs on its own thread.
- Can be conveniently implemented in Java using the `Executor`.
- **Non-blocking I/O**, which allows multiple clients per thread but requires more development time.
- Implemented in Java in the `java.nio` package.
- Less efficient than multi-threaded servers on modern machines.

Today's lecture

Today's lecture will focus on **network security**:

- Secure network communication.
- Private and public **encryption** keys.
- **Authentication** and certificates.
- Java implementation: The `SSLSocket` class.

This is only intended as an overview; many of you will take the Level 3 module *COMP3911 Secure Computing*, which will cover this material in more detail.

Where does network security happen?

Want security at multiple levels:

- Rapidly introduce new security at the Application layer.
- Network layer security broad, but cannot authenticate users.

Layer	Examples of Security protocols
Application	**PGP** = Pretty Good Privacy
Transport	**SSL** = Secure Socket Layer for TCP connections[1]
Network	**VPN** = Virtual Private Networks using **IPsec**
Link	**WEP** = Wired Equivalent Privacy for Wi-Fi
Physical	**Quantum communication** *(incoming)*.

[1] SSL really resides **between** the application and transport layers, but appears as a transport layer protocol to applications — see later in lecture.

Core concepts

The standard `Socket`-based communication in Java is fundamentally **insecure**.

Network traffic is unencrypted and can be intercepted.

- If messages were encrypted, would not be easy to read even if intercepted.
- The **encryption problem**.

If we write a server and publish the protocol, anyone can write a client that connects to the server.

- For instance, logging in to a server.
- The **authentication problem**.

Security concepts

Need to address two aspects:

Encryption:

- Traffic is 'secure' even if intercepted.

Authentication:

- Client can **trust** they are connected to the server they think they are — **server authentication**.
- We will not consider client authentication here.

Together they make an acceptable degree of security, although no system is totally secure.

Principles of encryption

- Start with a plain text message M.
- Assume[1] an **encryption algorithm** E exists such that it encrypts M into **cypher text** form $C = E(M)$.
- Further assume that a corresponding **decryption algorithm** D exists such that $M = D(C)$.

The encryption/decryption algorithms E and D are **known** (*i.e.* published), so we must combine their use with a secret **key** such that the effect of E and D cannot be predicted **without that key**.

[1] The *design* of these algorithms is beyond this module, but some of you may take the optional Level 3 module *COMP3223 Cryptography*.

Symmetric key algorithms

Also known as **private key cryptography**

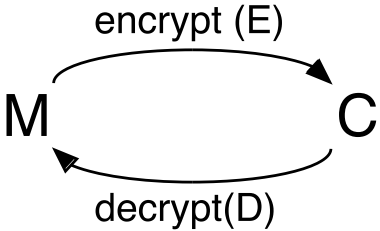
The 2 participants in the communication share knowledge of a **unique** key k:

- $C = E_k(M)$, $M = D_k(C)$.
- Both need the **same** key, hence **symmetric**.

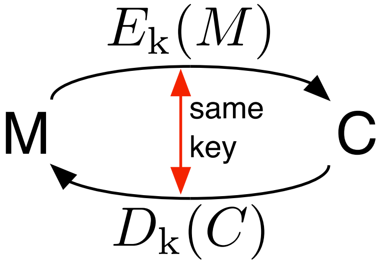
Standard algorithms include DES (Data Encryption Standard) and AES (Advanced Encryption Standard).

1. Client asks for k from server.
2. Server sends k.
3. Client encrypts message $C = E_k(M)$ and sends.
4. Server receives and decrypts $M = D_k(C)$.

No key:



With key:



Problems

The **key** must be communicated **in advance** such that both participants have knowledge of it.

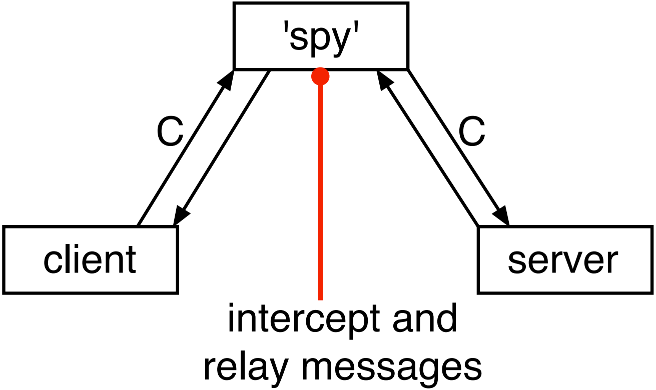
- If it is sent unencrypted, it could be **intercepted**.
- Would need to communicate by some other means.
- Could physically install key on each machine, but this would not allow network communications.

Anyone in possession of that key can read **all** communication.

- They could **relay** the messages to remain undetected[1].

[1] Quantum communication would detect such an interception.

$$M = D_{\mathrm{k}}(C)$$



Asymmetric key algorithms

Also known as **public key cryptography**

- Each participant has a private/public key **pair** k_{pvt}, k_{pub}.
- To send a message, ask for the recipient's **public** k_{pub}.
- Use this to encrypt the message: $C = E_{k_{pub}}(M)$.
- The recipient uses their key **pair** to decrypt: $M = D^{k_{pvt}}_{k_{pub}}(C)$.

This way, even if C is intercepted, it cannot be read because **both keys are required for decryption** — and the recipient never communicates the private key k_{pvt}.

For this to work, k_{pvt} and k_{pub} need to share a special relationship, but k_{pvt} cannot be inferred from k_{pub}. Standard implementation is RSA (<u>R</u>ivest-<u>S</u>hamir-<u>Adleman</u>).

In practice

- Symmetric key algorithms are relatively **efficient** in code.
- Asymmetric algorithms are **compute intensive**.

A common compromise is:

- Use an **asymmetric** algorithm to send a (short) **symmetric key**.
- Known as a **session key**.
- Use this key with a **symmetric** algorithm to encrypt and decrypt the (long) communication.

Since the key is secure, the symmetric algorithms are secure.

Man-in-the-middle attacks

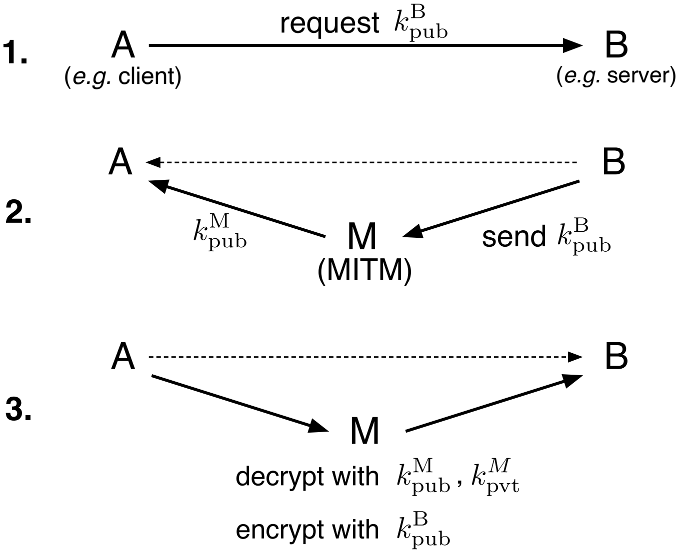
Since k_{pub} is public, **anyone** can use it to encrypt a message.

- The receiver does not know **who** is sending the message.

A possible scenario is:

1. A wants to send a message to B, using B's key k_{pub}^{B}.
2. However, a third party M intercepts k_{pub}^{B} and instead sends their key k_{pub}^{M} to A.
3. A encrypts their message using k_{pub}^{M}.
4. M intercepts the message, decrypts and reads it.
5. M then encrypts the message using k_{pub}^{B} and relays it to B.

This is known as an MITM (Man In The Middle) attack.


1.
request $k_{\mathrm{pub}}^{\mathrm{B}}$
A
(*e.g.* client)
B
(*e.g.* server)
2.
A
B
$k_{\mathrm{pub}}^{\mathrm{M}}$
M
(MITM)
send $k_{\mathrm{pub}}^{\mathrm{B}}$
3.
A
B
M
decrypt with $k_{\mathrm{pub}}^{\mathrm{M}}, k_{\mathrm{pvt}}^{M}$
encrypt with $k_{\mathrm{pub}}^{\mathrm{B}}$

Authentication

Adds **trust** to the transmitted public key.

The server sends a **certificate** containing its public key k^{S}_{pub} to any new client (superscript S for 'server').

- The certificate guarantees it came from the server.
- The client can check this.
- The client then sends k^{C}_{pub} for communication (C for 'client').

The certificate is **signed** by an approved **certification authority**.

- Very hard (but not impossible) to fake this.

Java uses standardised `X.509` certificates.

Obtaining a certificate

Commercial entities buy a certificate from a **certification authority**.

- *e.g.* VeriSign for SSL (Secure Sockets Layer).

Can generate and manage certificates using the `keytool` utility.

- Part of the standard Java distribution `Java SE`.
- Require certificates to implement our own secure communication.

Security and Java

Java provides the following packages for secure communication:

Java Secure Socket Extension (JSSE):

- `javax.net.*`
- Includes Secure Sockets Layer (SSL) as `javax.net.ssl.*`
- Also includes the more recent Transport Layer Security (TLS).

These protocols lie **inbetween** the Application and Transport layers.

- Technically breaks the 5-layer TCP/IP protocol (although not the 7-layer OSI protocol; see Lecture 2).

The SSLSocket class

An `SSLSocket` is constructed by an `SSLSocketFactory`.

- Use of factory (rather than a constructor) allows additional security (*e.g.* authentication).

Plain `Socket` with added layers of security protection:

```
public class SSLSocket extends Socket
```

Once created they act like a plain `Socket`.

Example:

```
SSLSocketFactory factory =
   (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) factory.createSocket(host,port);
```

Class details

Code on Minerva: `ListCipherSuites.java`. May not work on all systems[1].

By choosing different factories, can choose *e.g.* different methods and algorithms for authorisation or encryption.

`getDefault()` returns a factory class for **server-side authentication** with encrypted communication[1].

- Alternatives can be found using the `getSupportedCipherSuites()` method of `SSLSocketFactory`.
- `getEnabledCipherSuites()` gives those that are allowed.
- Needs to be negotiated by client and server.

Network communication functionality is inherited from the `Socket`.

[1] Works on `feng-linux.leeds.ac.uk` as of Nov. 2024.

The `SSLServerSocket` class

Plain `ServerSocket` with added layers of security protection:

```
public class SSLServerSocket extends ServerSocket
```

Once created they act like a plain `ServerSocket`, *i.e.* they inherit network functionality from `ServerSocket`.

Any client that wishes to connect **must** follow the server's security protocol.

- Part of the server protocol.
- Avoids risk of malicious client deliberately requesting weak security.

Echo server code fragment

Code on Minerva: `EchoServer.java`. Need set-up with `keytool`[1].

```
private void runServer() {
  try {
    sslsocket = (SSLSocket) sslserversocket.accept();
    BufferedReader bRead = new BufferedReader(
                            new InputStreamReader(
                             sslsocket.getInputStream()));
    String string = null;
    while( (string=bRead())!=null ) {
      System.out.println( string );
      System.out.flush();
    }
    sslsocket.close();
    sslserversocket.close();
  }
  catch( IOException ex ) { ... }
}
```

[1] See Worksheet 3 Question 2. Has been tested on `feng-linux.leeds.ac.uk`.

Echo client code fragment

Code on Minerva: EchoClient.java

As for the server, minimal changes to the (insecure) client code.

```
private void runClient() {
  try {
    BufferedReader bIn = new BufferedReader(
                          new InputStreamReader(
                           System.in ) );
    BufferedWriter bOut = new BufferedWriter(
                           new OutputStreamWriter(
                            sslsocket.getOutputStream() ) );
    String string = null;
    while( (string=bIn())!=null ) {
      bOut( string + '\n' );
      bOut();
    }
    sslsocket.close();
  }
  catch( IOException ex ) { ... }
}
```

The text sent by `bOut.write()` is now **encrypted**.

Overview and next lecture

Today we have briefly looked at network security:

- **Symmetric** key algorithms (**private** key only).
- **Asymmetric** key algorithms (private and **public** keys).
- The need for **authentication**.
- `SSLSocket` in `javax.net`.

So far all of the Java we have done has used **TCP**.

- Next time we will look at the other common transport layer protocol, **UDP**.