Advanced Sockets

Lecture 4 (B)

Secure Sockets Layer (SSL)

- A family of protocols layered over TCP and the **Sockets** API which provides:
  - Authentication
  - Integrity
  - Privacy services

NOTE:

RFC2246 defines an Internet standard called the Transport Security Protocol (TLS). As this was developed from earlier specifications by Netscape of a protocol called Secure Sockets Layer (SSL) the entire topic is often generically referred to as *secure sockets* or SSL.

Protocols in SSL/TLS

- **Record** protocol
- **Handshake** protocol
- **Session resumption** (optional feature)

Record Protocol

- Lowest level
- Provides cryptographic connection security which is **private** and **reliable**

Record Protocol

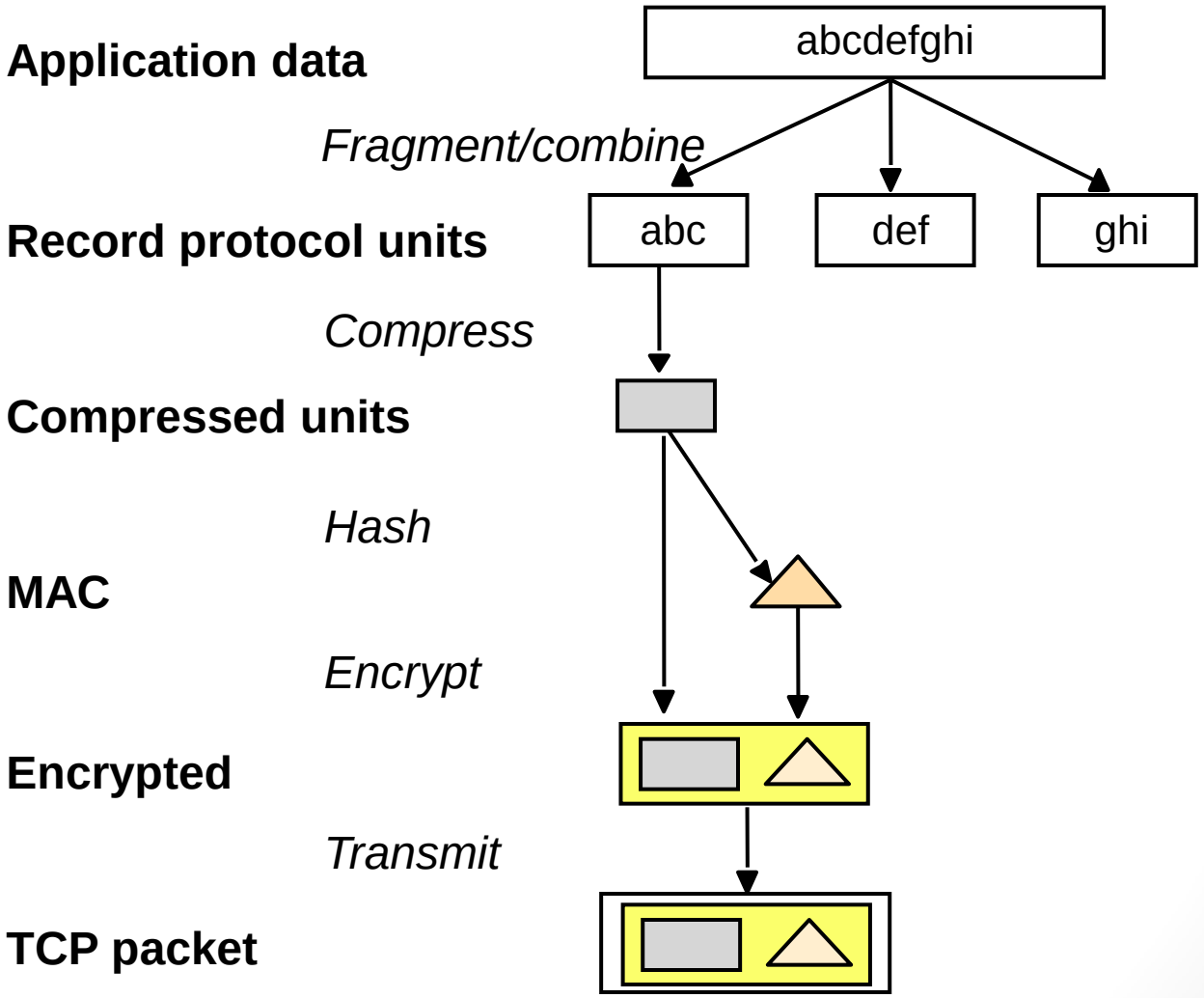
- **Privacy**
  - Provided by data encryption via one of a number of symmetric cryptographic techniques such as RSA, DES, RC4, etc
  - Encryption keys are generated uniquely for each connection, based on a secret negotiated by another protocol (such as the Handshake Protocol).

- **Reliability**
  - Provided via a message integrity check using a keyed Message Authentication Code (MAC)
  - Various secure hash functions (e.g. MD5, SHA, etc.) can be used for MAC computations

Record Protocol

- Used for encapsulation of higher level protocols, including **Handshake Protocol** & whatever application protocol is being transported.

- The operating environment of the record protocol consists of:
  - A compression algorithm
  - An encryption algorithm
  - A MAC algorithm.


Application data
abcdefghi
Fragment/combine
Record protocol units
abc
def
ghi
Compress
Compressed units
Hash
MAC
Encrypt
Encrypted
Transmit
TCP packet

Handshake Protocol

- Allows the server and client to authenticate each other and to negotiate an encryption algorithm and secret cryptographic keys before the application protocol transmits or receives its first byte of data.

- The Handshake Protocol is itself secure, having the properties of:
  - Authentication,
  - Privacy
  - Integrity:

Handshake Protocol

- Authentication
  - The peer's identity can be authenticated using asymmetric (public key) cryptography (e.g. RSA, DSS, etc.).
  - This authentication can be made optional, but it is generally required for at least one of the peers

Handshake Protocol

- Privacy
  - The negotiation of a shared secret is private: the negotiated secret is unavailable to eavesdroppers (passive attackers)
  - Provided that at least one endpoint of the connection is authenticated, the secret cannot be obtained even by an attacker who can place himself in the middle of the connection (an active attacker).

Handshake Protocol

- Integrity
  - The negotiation messages have integrity: no attacker can modify the negotiation communications without being detected by the parties to the communication.

Handshake Protocol

- The handshake can be initiated manually with the method:

```
class SSLSocket
{
  void startHandshake() throws IOException;
}
```

Handshake Protocol

- Regardless of whether the handshake is initiated manually or automatically, there are 5 possibilities:

a) No handshake has ever been performed between the peers. In this case the **startHandshake()** method is synchronous, and returns when the initial handshake is complete or throws an **IOException**. This handshake establishes thesession with its cipher suite and peer identities, as well as the cipher keys.

b) A handshake has been performed between the same peers; neither the client or the server has invalidated the session that resulted; and session resumption is supported by the JSSE implementation. In this case, the **startHandshake ()**method is asynchronous and returns immediately. If data has already been sent on the connection, it continues to flow during the handshake. This handshake only establishes new cipher keys for the connection.

Handshake Protocol

c) The session has been invalidated and another valid session is available for resumption. If data has already been sent on the connection, it continues to flow during the handshake. This handshake only resumes an existing session with new cipher keys for the connection. In this case the **startHandshake()** method is asynchronous and returns immediately. This case is not significanty different from case (b).

d) The session has been invalidated, no other valid sessions are available (or session resumption is not supported), and session creation has been disabled. In this case the handshake establishes a new session as in case (a); the handshake is asynchronous as in case (b).

Handshake Protocol

e) The session has been invalidated, no other valid sessions are available (or session resumption is not supported), and session creation has been disabled. In this case the handshake attempts to establish a new session as in case (a); the handshake is asynchronous as in case (b); the attempt will fail and the next read or write will throw an **SSLException**.

Sessions

- The handshake protocol results in a **session**, which establishes security parameters for use by the Record Layer when protecting application data

- It is a collection of security parameters agreed between two peers — 2 parties to a conversation — as a result of a handshake.

- It includes:
  - The peer identities (if established)
  - The encryption and compression methods to be used between the peers
  - The private and public keys and master secrets used by these methods.

Sessions

- An SSL session can outlive the connection which produced it, and can be *resumed* by a subsequent or simultaneous connection between the same two peers.

- This *session resumption* feature of SSL allows the security parameters associated with a single secure session to be used for multiple conversations (connections) *sequentially* or *concurrently*, i.e. for connections formed one after the other or at the same time

Secure Client Sockets

- Using an encrypted SSL socket to talk to an existing secure server is truly straightforward

- Rather than constructing a **java.net.Socket** object with a constructor, you get one from a **javax.net.ssl.SSLSocketFactory** using its **createSocket()** method.
  - **SSLSocketFactory** is an *abstract class* that follows the abstract factory design pattern:

```
public abstract class SSLSocketFactory
extends SocketFactory
```

Secure Client Sockets

- Since the SSLFactorySocket class is itself abstract, you get an instance of it by invoking the static **SSLSocketFactory.getDefault()** method:

```
public static SocketFactory
getDefault( ) throws
InstantiationException
```

- This either returns an instance of **SSLSocketFactory** or throws an **InstantiationException** if no concrete subclass can be found.

Secure Client Sockets

- Once you have a reference to the factory, use one of these five overloaded **createSocket( )** methods to build an **SSLSocket**:
  1. `public abstract Socket createSocket(String host, int port) throws IOException, UnknownHostException`

  2. `public abstract Socket createSocket(InetAddress host, int port) throws IOException`

Secure Client Sockets

- (continued)

3. `public abstract Socket createSocket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`

4. `public abstract Socket createSocket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`

5. `public abstract Socket createSocket(Socket proxy, String host, int port, boolean autoClose) throws IOException`

Secure Server Sockets

- Instances of the **javax.net.SSLServerSocket** class:
  - `public abstract class SSLServerSocket extends ServerSocket`
- Like **SSLSocket**, all the constructors in this class are protected.
- Like **SSLSocket**, instances of **SSLServerSocket** are created by an abstract factory class, **javax.net.SSLServerSocketFactory**:
  - `public abstract class SSLServerSocketFactory extends ServerSocketFactory`
- Also like **SSLSocketFactory**, an instance of **SSLServerSocketFactory** is returned by a static **SSLServerSocketFactory.getDefault( )** method:
  - `public static ServerSocketFactory getDefault()`

Secure Server Sockets

- And like **SSLSocketFactory**, **SSLServerSocketFactory** has three overloaded **createServerSocket( )** methods that return instances of **SSLServerSocket** and are easily understood by analogy with the **java.net.ServerSocket** constructors:
  - `public abstract ServerSocket createServerSocket(int port) throws IOException`
  - `public abstract ServerSocket createServerSocket(int port, int queueLength) throws IOException`
  - `public abstract ServerSocket createServerSocket(int port, int queueLength, InetAddress interface) throws IOException`

Secure Server Sockets

- The factory that **SSLServerSocketFactory.getDefault( )** returns generally only supports server authentication.
  - It does not support encryption.

- To get encryption as well, server-side secure sockets require more initialization and setup.

Implementing Secure Server Sockets

1. Generate public keys and certificates using *keytool*.
2. Pay money to have your certificates authenticated by a trusted third party such as Verisign.
3. Create an **SSLContext** for the algorithm you'll use.
4. Create a **TrustManagerFactory** for the source of certificate material you'll be using.
5. Create a **KeyManagerFactory** for the type of key material you'll be using.

Implementing Secure Server Sockets (continued)

6. Create a **KeyStore** object for the key and certificate database. (Sun's default is *JKS*.)
7. Fill the **KeyStore** object with keys and certificates; for instance, by loading them from the filesystem using the pass phrase they're encrypted with.
8. Initialize the **KeyManagerFactory** with the **KeyStore** and its pass phrase.
9. Initialize the context with the necessary key managers from the **KeyManagerFactory**, trust managers from the **TrustManagerFactory**, and a source of randomness. (The last two can be null if you're willing to accept the defaults.)