私立東海大學資訊工程與科學系

碩士論文

指導教授：林 祝 興 博士

(Dr. Chu-Hsing Lin)

電子病歷分享系統中安全技術之設計與實作

Design and Implementation of Security Technologies for an Electronic Health Records Sharing System

研 究 生：劉 廷 楷 撰

(Ting-Kai Liu)

中 華 民 國 九 十 二 年 六 月

摘　　要

從病歷開始電子化以來，如今已十分普及地實行於各醫院之間。然而，其應用範圍卻僅僅侷限於各醫院內部。想要將電子病歷在醫院與醫院之間作交換的話，首先得克服各醫院間醫療資訊系統的不同以及資料庫無法整合的問題。因此，雖然病歷已經電子化，但還是無法達到病歷分享的目標。追究問題所在主要是因為雖然電子化病歷交換的標準已定，但各醫院內原有的醫療資訊系統在支援 HL7 的標準上，有其實作上的困難點。

因此，本論文的目標是在於設計一套適合於電子病歷分享系統的安全技術，包括 Smart Card、公開金鑰基礎架構(Public Key Infrastructure, PKI)、憑證中心、公開金鑰電子憑證(Public key certificate)、電子簽章以及 SSL，並且實作電子簽章模組和 SSL 安全連線模組，進而建立一套快速且安全的電子病歷分享模式。

關鍵字：Smart Card、公開金鑰基礎架構(Public Key Infrastructure, PKI)、憑證中心、公開金鑰電子憑證(Public Key Certificate)、電子簽章、SSL(Secure Socket Layer)

Abstract

Since health records had been electrified, it has been commonly implemented in every hospital now. However, the application range of electrical health records are still limited within every hospital. If we would like to exchange electrical health records among the hospitals, we firstly should overcome the problems of different health information systems and unavailable integrated databases among the hospitals. Therefore, although health records are electrified, it still can not achieve the goal of health records which are shard. Investigating the main reason which the electrical health records can not be extended is that although the standard of the electrical health records has been made, the original health information system of every hospital which supports the standard of HL7 is difficult to be implemented.

Therefore, the target of this paper is to design a kind of security technology which fits for the electronic health records sharing system, including smart card, PKI, certificate authority, public key certificate, signature and SSL. And I implement the program module of digital signature and SSL to establish an efficient and secure electronic health records sharing system.

Keywords： Smart Card、PKI(Public Key Infrastructure)、Certificate Authority、Public Key Certificate、Signature、SSL(Secure Socket Layer)

目　　錄

圖表目錄

第壹章 序論

第一節 研究背景

隨著資訊技術的發達與網際網路的盛行，在醫療方面，現在國內多數醫療院所在醫療資訊電腦化和病歷電子化方面，都已幾乎轉換完畢，並且遵循著電子病歷分享系統，將醫療院所各自的電子病歷格式轉換成共同的格式存放在另一個專為電子病歷分享系統建立的電子病歷資料庫裡，同時再透過 HL7 這醫療資訊傳遞的標準協定互相交換彼此的電子病歷。

在這電子病歷分享系統中，很明顯地面臨到如何保護放到網際網路上的電子病歷的安全問題：一、如何確保下載病歷的人和被下載的病歷資料是合法的？二、如何確保在電子病歷下載過程中不被外界所竊取？三、如何確定下載到的病歷是沒有被竄改過的？

公開金鑰電子憑證(Public Key Certificate)由於是透過具有公信力且可依賴的第三者，憑證中心(Cerfitication Authority, CA)，所發放的，目的在提供使用者或伺服器一張電子身分證，可以給予在網際網路上資料交換前的雙方確認對方的身份是合法且可信任的。電子簽章提供了從網際網路收到的資料是否被竄改的依據，通常資料提供者會連同原資料和簽章資料一同送給資料接收者，假使收接者懷疑資料已被竄改，可以經由比對解密過後的電子簽章連同雜湊過後的原資料作比

對，如果比對結果相同，便可確定資料沒有被竄改過，此外，隨著電子簽章法在民國九十一年四月一日的通過，更確立電子病歷的法律地位。SSL(Secure Socket Layer)加密機制提供資料傳遞雙方一個安全的連線環境，所有要傳遞的資料會被加密過，確保資料的隱密性。

於是，如何設計並實作一套適用於電子病歷分享系統的安全技術，讓醫療院所可以在轉換其院內的電子病歷成共同的格式後，透過共同的訊息傳遞標準安全地交換彼此的資訊成為本論文的研究主題重心。

第二節 論文架構

本文除了本章序論外，第二章將提出相關的資訊安全技術研究，包括公開金鑰基礎架構(PKI, Public Key Infrastructure)、X.509 認證服務、電子簽章和安全通道層(SSL, Secure Socket Layer)等。第三章說明本論文所提的電子病歷分享模式的架構、實施方法及進行步驟。第四章針對架構中用到的安全機制進行說明。第五章為結論，探討在這個模式裡安全模組部分實作的經驗及心得，以及未來還可以加強的安全機制。

第貳章　相關安全技術研究

第一節 公開金鑰基礎架構

在電子化的身分認證系統中，通常面臨到一個共同的問題，便是身分認證的關係只建立在使用者與特定的電腦系統之間，而無法將這種關係延伸到更多的使用者團體與更廣的網路系統；也就是說，這樣的身分認證系統並無法成為真正的公共身分認證系統。

公開金鑰基礎架構提供了公共的身分認證系統，目的在維護每個使用者的憑證的正確性，也就是使用者公開金鑰的正確性，能夠證明此公開金鑰是屬於某一位特定使用者所擁有。

公開金鑰基礎架構由下列幾個主要角色所組成(圖 2-1)：

- **終點實體(End Entity)**：使用憑證的的主要對象，終點實體可視為一般使用者、團體、公司行號等。
- **憑證中心(CA, Certification Authority)**：可信任的公正第三者，負責憑證、憑證廢止清冊之發行與管理等工作。
- **註冊中心(RA, Registration Authority)**：位於憑證中心之前端，終端實體可透過註冊中心申請憑證。
- **憑證/憑證廢止清冊資料庫(Certificate/CRL Repository)**：公開金鑰基礎架構中的資料儲存體，負責儲放憑證、憑證廢止清冊之資料。

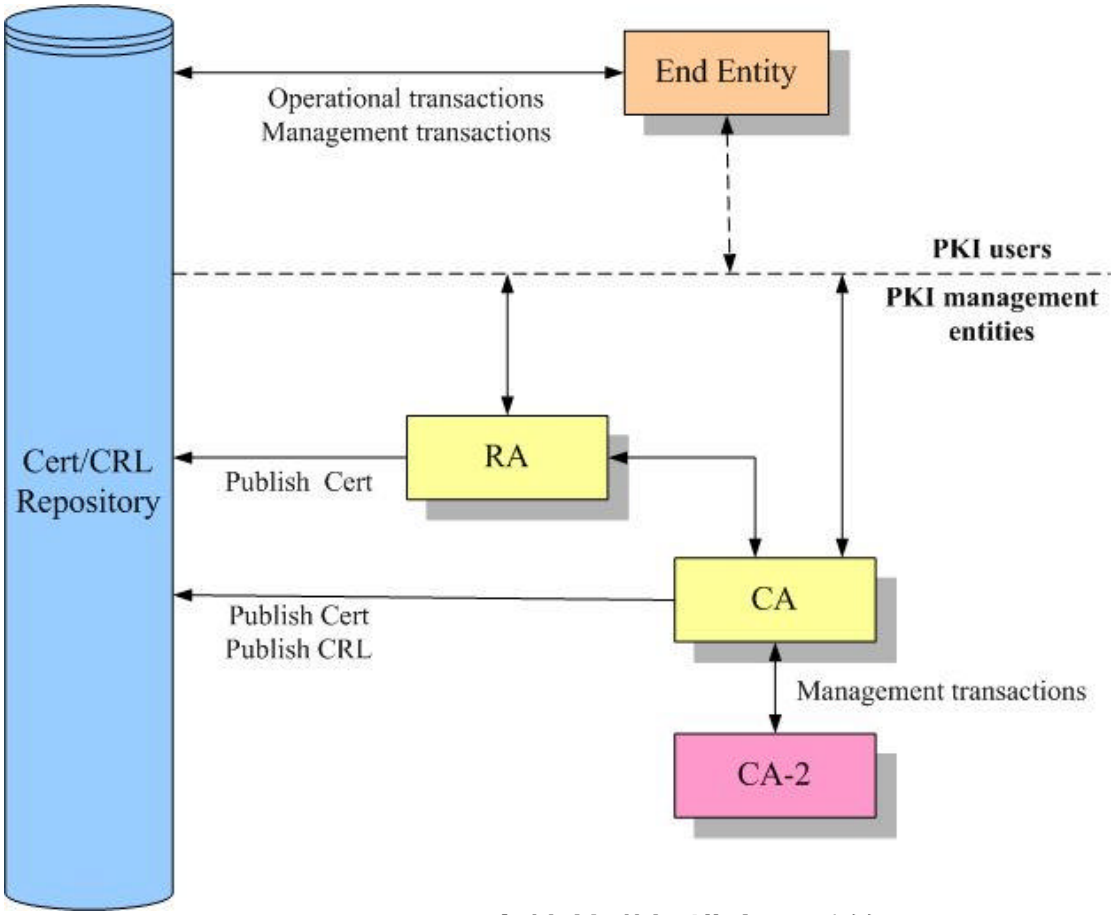

圖 2-1 公開金鑰基礎架構之憑證管理

一、憑證中心

憑證中心是一種組織，在公開金鑰基礎架構中扮演的是可信任的公正第三者，目的在對個人及機關團體提供認證及憑證簽發管理等服務，以建立具有機密性(Confidentiality)**、鑑別**(Authentication)**、完整性**(Integrity)**、不可否認性**(Non-repudiation)**、存取控制**(Access control)**及可用性**(Availability)**的資訊通訊安全環境與機制。主要工作是負責憑證與憑證廢止清冊之發行、憑證與公開金鑰的管理以及處理與其他認證中心相互之間的信任關係。**

然而，在實際的運作上，憑證中心不可能完全地處理每一個使用者申請憑證的作業，因為憑證中心不可能驗證所有申請憑證之使用者的身分，所以必須透過前端的註冊中心來當協助處理身分驗證的機制，再交由憑證中心發行憑證；如果註冊中心本身是一個可自行發行憑證的組織單位，亦可以由註冊中心代為發行憑證。註冊中心的角色並不一定為一特定的人或單位來執行，只要能夠確認使用者身分資料的單位，都可以是註冊中心。

在建置營運憑證中心時，須依憑證中心之營運政策及策略，制訂憑證政策(Certification Policy，CP)與憑證實作準則(Certificate Practice Statement，CPS) ，規範其運作規定與作法，一方面讓用戶瞭解在使用上的作業規定，另一方面則藉此表明其在安全性及公正性上的可信賴度。

二、憑證發行機制

憑證的發行機制可分成兩種，基本驗證機制(basic authenticated scheme)與集中管理機制(centralized scheme)[8]。

■ 基本驗證機制(basic authenticated scheme)：

由申請憑證之使用者產生金鑰對(key pair)及憑證要求(certificate request)，將憑證要求以憑證中心之 IAK(Initial Authentication Key)加密傳回憑證中心進行驗證，再依憑證要求產

生使用者之憑證並發行(圖 2-2)。

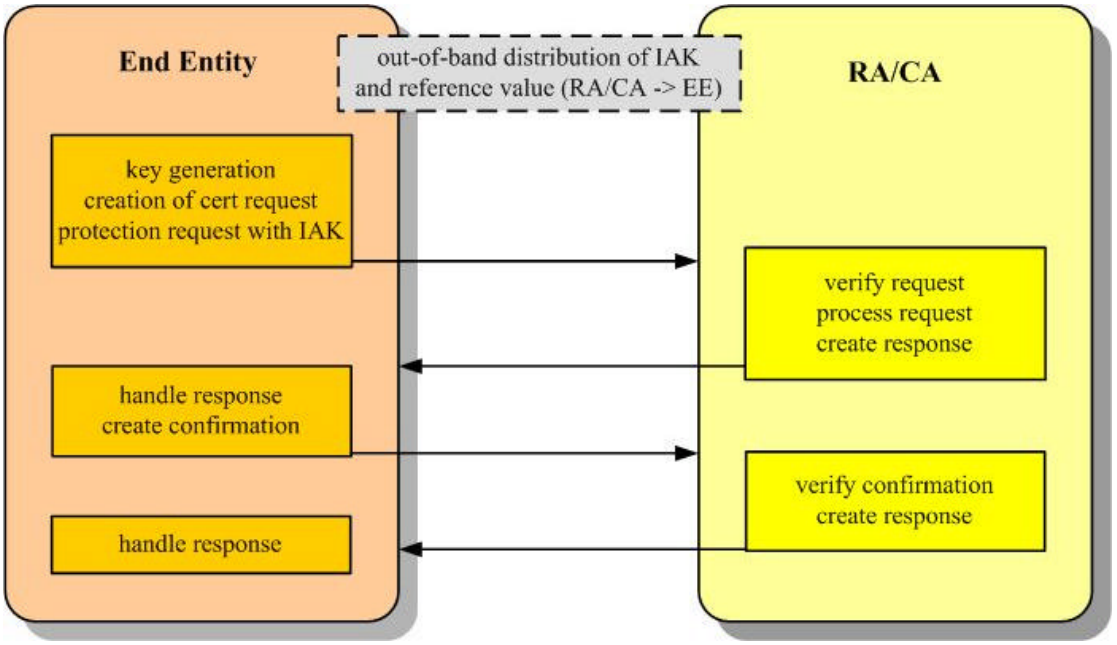

圖 2-2 憑證發行－基本驗證機制

■ 集中管理機制(centralized scheme)：

使用者將 PSE(Personal Security Environment)資料送至憑證中心，由憑證中心產生金鑰對及憑證，並將憑證發行。

第二節 X.509

電子化作業在網際網路上的應用日益廣泛, 同時衍生原本傳統人工作業上所沒有的資訊安全問題，其中，網路上的身分認證(Authentication)機制便是一項重要課題。為了在開放式網路架構下認證遠端使用者的身分，ITU (International Telecommunication Union)於 1988 年提出 ”開放性系統互連－目錄服務：認證架構 ”(Open System Interconnection－The Directory：

Authentication Framework) X.509，之後此標準為國際標準組織 ISO(International Standard Organization)所採用為 ISO 9594-8 認證標準 目前幾乎所有的公開金鑰建設(PKI)標準都是架構在 X.509 此標準所發展出來的。

一、X.509

在 ITU 提出之 X.509 規格中，對於認證(Authentication)提出兩種不同安全度的認證層級：「簡單認證」與「強認證」；X.509 規格中並描述了公開金鑰憑證(Public Key Certificate)格式[10]、憑證管理[11]、憑證路徑、目錄資料結構及金鑰產生與管理[12]等，其中並包含憑證中心間交互承認的憑證之儲存，以減少憑證驗證時必須從目錄服務中獲得憑證資訊。X.509 規格內容可歸納包含下列項目：

- **簡單認證程序**(Simple Authentication Procedure)：

  此部份所定義的驗證程序是使用一般最常見的密碼(password)認證的技術來辨識通訊的對方。

- **強認證程序**(Strong Authentication Procedure)：

  此部份內容，提出一個高安全度的身分認證機制，就是使用公開金鑰密碼學的技術，來辨識通訊的對方。在 X.509 提出的認證架構，並非要建立一個通用的認證架構，整個認證協定只有描述資料項。

■ **金鑰及憑證管理**(Key and Certificate Management)：

因為強認證程序需使用公開金鑰密碼系統的特性來進行認證工作，因此，此部份針對金鑰與憑證的管理與正確性做一重點摘要，並定義了憑證廢止清冊(Certificate Revocation List，CRL)**的內容。**

■ **憑證擴充及憑證廢止清冊**(Certificate Extensions and CRL)：

由於 1988 **年版的** X.509 **中對憑證及憑證廢止清冊的定義並不是很完整，所以在** 1995 **年針對這些問題，提出了** X.509 **修正案，對於這兩部分做一些修正與補充。**

二、X.509 憑證

1. **在** X.509 **規格中所定義的憑證**(Certificate)**必須要符合下列兩點特性：**

■ **任何可由憑證中心取得公開金鑰的使用者，都可以找到已經通過認證的公開金鑰。**

■ **除了憑證中心以外，任何人修改憑證的動作都會被偵測、察覺。**

2. X.509 **憑證除了符合上述兩個特性外，憑證內容必須包含以下資訊欄位**(**圖** 2-4)：

■ **版本(Version)：此憑證發行所依循的版本。**

■ **序號(Serial number)：憑證中心對此憑證所發予唯一的序號。**

- **簽章演算法(Signature algorithm identifier)：憑證中心發行此憑證使用之簽章演算法，用以驗證簽章時使用。**
- **發行者名稱(Issuer name)：發與此憑證之憑證中心名稱。**
- **有效期限(Period of validity)：憑證之有效期限，包含起始日期與終止日期。**
- **主旨名稱(Subject name)：此憑證之擁有人之使用者名稱。**
- **公開金鑰資訊(Public key information)：包含此憑證之公開金鑰與公開金鑰相關資訊。**
- **簽章(Signature)：憑證中心以其私密金鑰對此憑證之簽章。**

了解上述符合 X.509 規格之憑證需包含的內容資訊後, 憑證可以下列格式表示如下：

CA<<A>> = CA{V, SN, AI, CA, T_A, A, Ap, (UCA), (UA)}

其中各個符號所代表的意義如下：

A ：使用者名稱。

V ：憑證版本。

SN：憑證序號。

AI：簽章演算法。

CA：憑證中心名稱。

T_A：憑證之有效期限。

A_p：憑證之公鑰，包含公鑰相關資訊。

(UCA)：憑證中心識別碼(選擇性使用)。

(UA) ：使用者識別碼(選擇性使用)。

Y<<X>>：憑證中心 Y 核發使用者 X 之憑證

Y{I}：憑證中心 Y 對訊息 I 所做的簽章，其中包含訊息 I 及簽章。

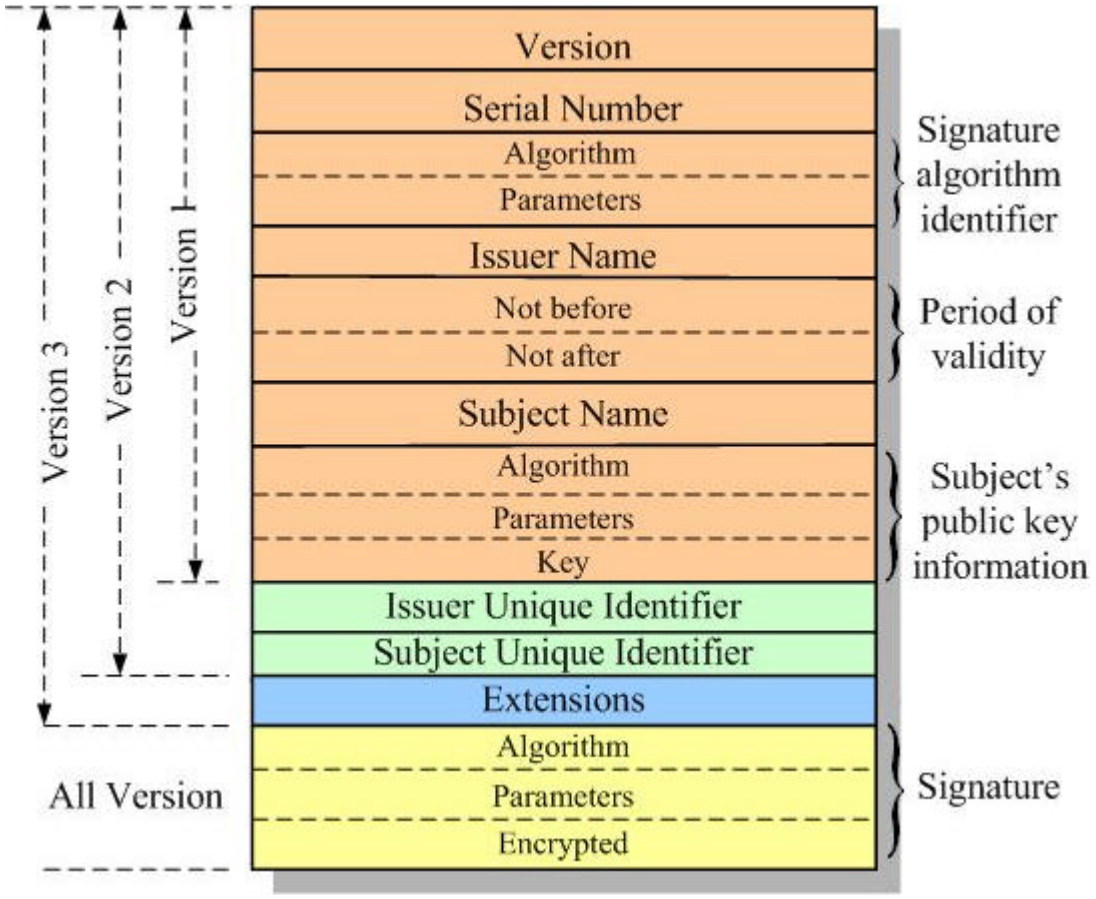

圖 2-3 X.509v3 憑證格式

憑證中心利用自己的私密金鑰來對憑證做簽章, 如果使用者知道對應的公開金鑰，就可以確認此憑證是否確實為該憑證中心所簽署的憑證。

三、X.509 憑證廢止清冊

在憑證內容中定義了有效期限，如信用卡一樣，在舊憑證快要過期之前，我們就要發出新的憑證。除此之外，在某些情況下，我們會在憑證到期之前就廢止它。可能原因如下：

- 使用者的私密金鑰被破解。
- 使用者不再由此憑證中心來認證。
- 憑證中心的憑證被破解。

因此，憑證中心必須維護一張清冊，清冊單記錄的是已經被廢止但是尚未過期的憑證；這些憑證可能是發給使用者或其他的憑證中心。此憑證廢止清冊(Certificate Revocation List)經由發行者的簽署，內容包含發行者的名稱、清單的產生日期、下一張憑證廢止清冊的預定核發日期，以及所有被廢止的憑證(圖 2-5)。廢止的憑證記錄中包含憑證的序號及廢止日期，因為序號對憑證中心而言是唯一的，所以利用序號就足以辨識這些被廢止的憑證。

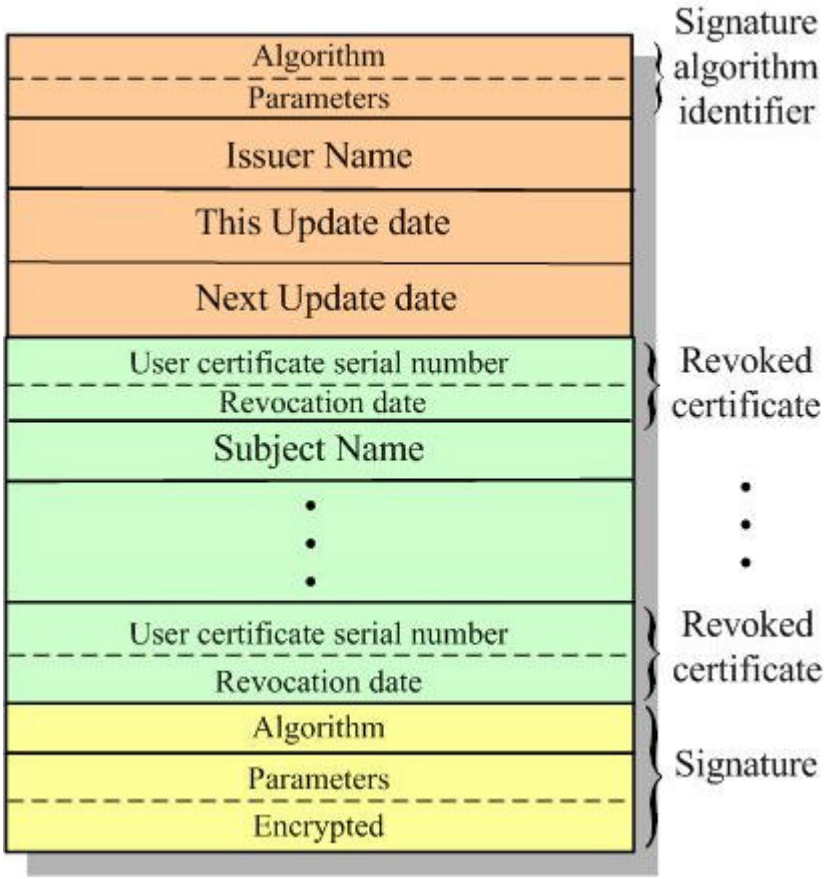

圖 2-4　X.509 憑證廢止清冊格式

四、X.509 v3 – 憑證/憑證廢止清冊擴充

由於 1988 年版的 X.509 規格中對憑證及憑證廢止清冊的定義並不是很完整，對後來的設計與實作角度來看，以 X.509 第二版的格式並無法傳送足夠所需要的資訊，所以在 1995 年針對這些問題，提出了 X.509 修正案，對於這兩部分做一些修正與補充。除了對上述問題提出解決方案外，為了避免因為增加固定格式的欄位而必須做版本的修改，修正的 X.509 第三版提出了選擇性的擴充資料欄位，每個擴充欄位項目包含了一個擴充識別碼、一個關鍵指標及一個擴充值。

X.509 v3 定義的憑證與憑證廢止清冊中的擴充項目，依功能可分為三類：金鑰與策略的資訊(key and policy information)、憑證擁有者與憑證發行者的屬性(certificate subject and certificate issuer attributes)，及憑證路徑的限制(certification path constraints)。

1. **金鑰與策略的資訊**
   - **金鑰識別碼**(Authority key identifier)
   - **使用者金鑰識別碼**(Subject key identifier)
   - **金鑰用途**(Key usage)
   - **私密金鑰使用期限**(Private-key usage period)
   - **憑證策略**(Certificate policies)
   - **策略對映**(Policy mapping)
2. **憑證擁有者與憑證發行者的屬性**
   - **使用者別名**(Subject alternative name)
   - **發行者別名**(Issuer alternative name)
   - **使用者目錄屬性**(Subject directory attributes)
3. **憑證路徑的限制**
   - **基本限制**(Basic constraints)
   - **名稱限制**(Name constraints)

- **策略限制**(Policy constraints)

五、憑證路徑驗證

利用憑證來進行身分確認的動作，如果所有使用者都採用同一個憑證中心的話，那他們就會同時信任這個憑證中心，因此也可以直接信任其他使用者的憑證。

當有很多使用者時，通常讓所有使用者採用同一個憑證中心的情況，並不是很實際的做法。在使用者的憑證為不同憑證中心所簽發的情況下，如何驗證對方憑證的正確與合法性，我們必須透過憑證路徑驗證來達到此一目的。在我們要驗證一張個別的憑證時，必須先建立其憑證路徑(Certificate Path)或憑證鏈(Certificate Chain)，也就是將該憑證與其關聯的憑證中心之憑證建立階層式架構，一直到我們可以信賴的第三公正單位或憑證中心。憑證路徑的驗證必須滿足下列條件(假設憑證路徑中共有 n 張憑證{1，… ，n})：

- **在**{1, …, n-1}**的憑證路徑中，所有的憑證** x**，其憑證的主旨**(subject)**必須是憑證** x+1 **的發行者名稱**(issuer)**。**
- **憑證** 1 **必須是一個可信賴的公正單位。**
- **憑證** n **必須是終端使用者之憑證(非可簽發憑證的憑證中心)。**
- **在**{1, …, n-1}**的憑證路徑中，所有憑證** x **必須都是在有效期限之間。**

憑證路徑之驗證演算步驟如下(圖 2-6)：

(1).初始化(initialization)

(2).基本憑證驗證處理(basic certificate processing)

(3).下一張憑證處理準備(preparation for next certificate)

(4).包裝(wrap-up)

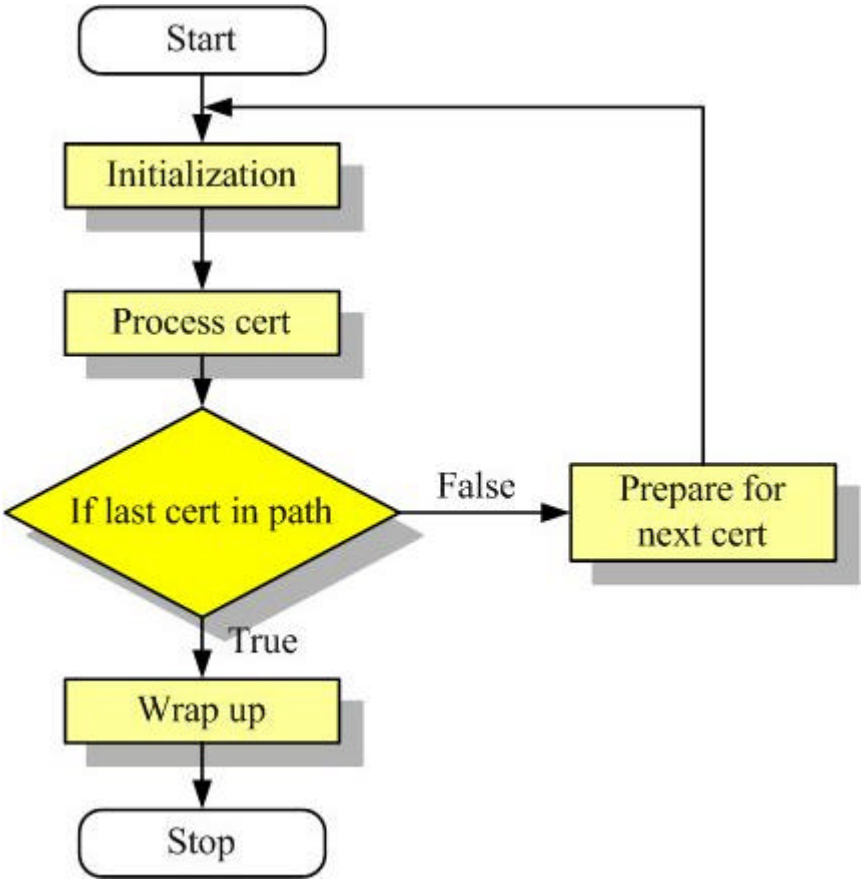

圖 2-5　憑證路徑之驗證演算步驟

六、憑證路徑驗證演算法

前一章節定義了憑證路徑的驗證，並且說明了驗證作業包含四個主要步驟：初始化、基本憑證驗證處理、下一張憑證處理準備、包裝，

此一章節針對這四個步驟，說明每一步驟的主要功能及細節。

1. **輸入**(Input)：

憑證路徑驗證演算法的包含了七個輸入值：

(1).**憑證路徑：憑證路徑長度為** n，**亦即內容包含** n **張憑證**(1~n)。

(2). T：**目前日期與時間。**

(3). user-initial-policy-set：**使用者憑證策略，若不使用憑證策略可放入一個** any-policy **值。**

(4).Trust anchor information：**憑證路徑中可信賴第三者(憑證中心)之資訊，包含發行者名稱**(issuer name)、**公開金鑰演算法**(public key algorithm)、**公開金鑰參數**(public key parameters)。

(5). initial-policy-mapping-inhibit：**指示可否使用** policy-mapping。

(6). initial-explicit-policy：**指示可否使用** explicit-policy。

(7). initial-any-policy-inhibit：**指示可否使用** any-policy。

2. **初始化**(Initialization)

初始化的步驟中，主要在建立憑證路徑驗證演算法所使用之變數；根據輸入的七個參數值，建立十一個狀態變數(state variables)：

- valid_policy_tree
- permitted_subtrees
- excluded_subtrees
- explicit_policy

- inhibit_any-policy
- policy_mapping
- working_public_key_algorithm
- working_public_key
- working_public_key_parameters
- working_issuer_name
- max_path_length

3. **基本憑證驗證處理** (Basic Certificate Processing)

完成初始化步驟後，開始針對 1~n **之憑證進行驗證處理，其中驗證步驟又可細分如下：**

(1).**驗證憑證之基本資料**

檢查及驗證憑證內容之基本資料的正確性, 驗證的內容包含下列作業：

- **簽章是否正確。**
- **憑證是否在有效期限內。**
- **憑證是否被列入憑證廢止清冊。**
- **發行者名稱是否合法。**
- **發行者識別碼是否合法。**

(2).**驗證主旨名稱**(subject name)

(3).**驗證憑證策略擴充**(certificate policies extension)

4. **下一張憑證處理準備** (preparation for next certificate)

在基本憑證驗證處理步驟完成後，如果基本憑證驗證處理的憑證並不是憑證路徑的最後一個(第 n **張憑證)，則需要在處理下一個憑證驗證作業之前，建立一些狀態變數。假設目前完成基本憑證驗證作業的為憑證** i (i=1, 2, 3, …. , n-1)**，下一張憑證驗證處理準備作業步驟如下：**

(1).**如果憑證** i **有** policy mapping extension**，驗證** any-policy **是否出現在** issuerDomainPolicy **或** subjectDomainPolicy **中。**

(2).**如果憑證** i **有** policy mapping extension**，則修改** valid_policy_tree**。**

(3).**將憑證** i **之主旨名稱**(subject name)**放入** working_issuer_name **變數中。**

(4).**將憑證** i **之公開金鑰放入** working_public_key **變數中。**

(5).**將憑證** i **之公開金鑰參數放入** working_public_key_parameters **變數中。**

(6).**將憑證** i**之公開金鑰演算法放入** working_public_key_algorithm **變數中。**

(7).**如果憑證** i **有** name constraints extension**，則修改相關之狀態變數**(permitted_subtrees、excluded_subtrees)**。**

(8).**如果發行者與主旨名稱不一致，則檢查**

- **如果** explicit_policy **值大於** 0，explicit_policy **值減** 1**。**
- **如果** policy_mapping **值大於** 0，policy_mapping **值減** 1**。**
- **如果** inhibit_any-policy **值大於** 0，inhibit_any-policy **值減** 1**。**

(9).**如果憑證** i **有** policy constraints extension**，則修改相關之狀態**

變數(explicit_policy、policy_mapping)。

(10).如果憑證 i 有 inhibitAnyPolicy extension，且其值小於 inhbit_any- policy，則將 inhbit_any-policy 值設為 inhibitAnyPolicy。

(11).驗證憑證 i 是否為一張 CA 的憑證。

(12).如果憑證 i 並非自己簽發(self-issued)，則檢查 max_path_length 值是否大於 0，若值大於 0，則將 max_path_length 值減 1。

(13).如果憑證 i 有 pathLengthConstraint，且其值小於 max_path_length，則將 max_path_length 值設為 pathLengthConstraint。

(14).如果憑證 i 有 key usage extension，將 keyCertSign 這個位元設定為 1。

(15).確認及處理憑證 i 中其他關鍵性的擴充參數。

5. 包裝 (Wrap-up)

完成憑證 i 到憑證 n-1 的基本憑證驗證處理與下一張憑證處理準備後，相同的，憑證 n(使用者憑證)也經基本憑證驗證處理，但完成後便進入包裝的步驟。包裝的過程如下：

(1).如果憑證 n 並非自行簽發(self-issued)，且 explicit_policy 值大於 0，則將 explicit_policy 值減 1。

(2).如果憑證 n 有 policy constraints extension 及 requireExplicitPolicy，且 requireExplicitPolicy 值為 0，則將 explicit_policy 設定為 0。

(3).將憑證之主旨名稱(subject name)放入 working_issuer_name 變數中。

(4).**將憑證之公開金鑰放入** working_public_key **變數中。**

(5).**將憑證之公開金鑰參數放入** working_public_key_parameters **變數中。**.

(6).**確認及處理憑證** n **中其他關鍵性的擴充參數。**

(7).**計算** valid_policy_tree **及** user-initial-policy-set**。**

6. **輸出** (Output)

輸出主要在表現整個憑證路徑驗證處理過程為成功或失敗。若是驗證失敗，則輸出驗證失敗的代表值及驗證失敗的警訊；若是驗證成功，除了輸出驗證成功的代表值外，同時輸出下列參數的最終值(處理憑證路徑中憑證 n **的值)：**

- valid_policy_tree
- working_public_key
- working_public_key_algorithm
- working_public_key_parameters

七、屬性憑證

目前 ITU **根據** X.509v3 **之規格，針對存取控制擬定了屬性憑證**(Attribute Certificate)**標準，並公佈為網際網路草案**(inernet draft)**，目的在取代現有的** X.509v3 **規格。屬性憑證是從使用者的公鑰憑證(為了方便區分憑證類型，我們將上述的** X.509 **憑證稱為公鑰憑證)衍生而來的一種分散式結構；公鑰憑證主要用途是包裝主體的公鑰，而屬性憑證則用來存放主體的相關屬性，因此屬性憑證並不包含主體的公**

鑰。一個公鑰憑證的主體可能會有一個以上的屬性憑證，而這些屬性憑證與其公鑰憑證均具有關聯性。公鑰憑證和屬性憑證並不一定要由相同的機構所發行，否則當憑證的發行與管理工作頻繁，將造成發行機構的負擔。在不同的發行機構的環境下，憑證中心發行的公鑰憑證和屬性憑證中心(Attribute Authority, AA)發行的屬性憑證會以不同的私鑰來做簽章。如果發行公鑰憑證的 CA 和發行屬性憑證的 AA 是在同一個發行機構下，最好將此發行機構視為屬性憑證而非標示公鑰憑證。屬性憑證內容如下：

- 版本(Version)：屬性憑證的版本。
- 持有者(Holder)：包含公鑰憑證發行者或公鑰憑證之序號、主體名稱與物件摘要。
- 簽章演算法(Signature Algorithm Identifier)：屬性憑證中心簽發此屬性憑證之演算法。
- 屬性憑證序號(Serial Number)：屬性憑證中心發行此屬性憑證的唯一序號。
- 有效期限(Period of validity)：包含起始日期及終止日期。
- 發行者識別(Issuer Identifier)：屬性憑證中心之識別名稱。
- 屬性(Attributes)：主體的屬性值，可包含一個以上的屬性值。
- 屬性憑證擴充(Extensions)

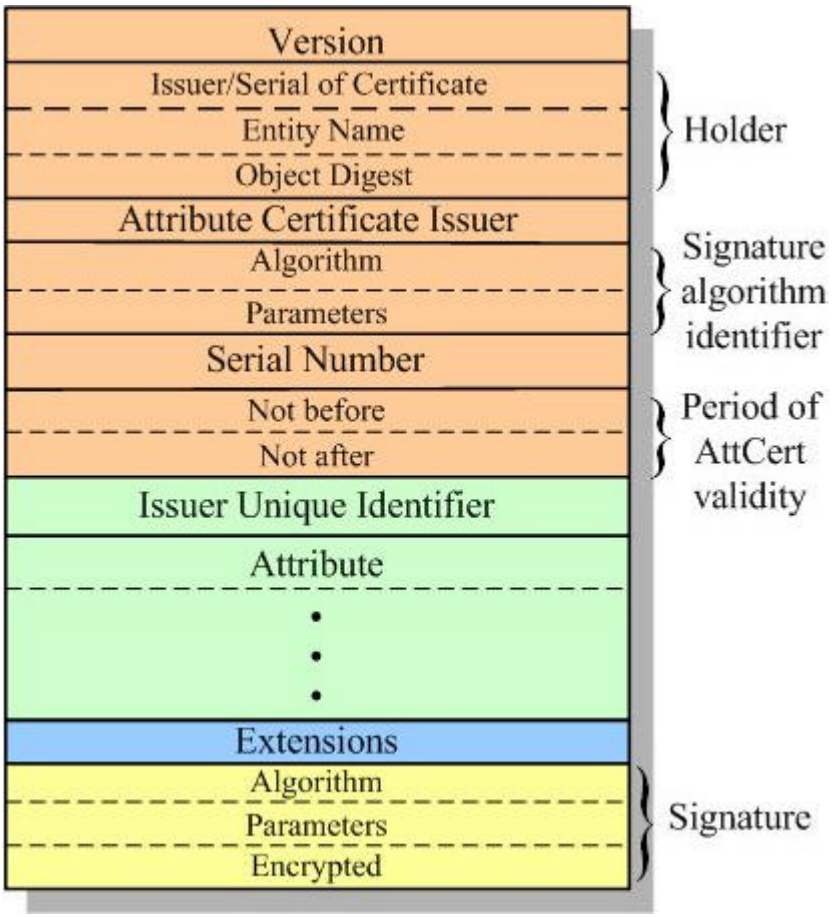

圖 2-6 屬性憑證格式

第三節 數位簽章

早在 1970 **年，銀行間便開始利用他們原有的金融資訊網路進行電子資金轉換（**Electronic Funds Transfer，EFT**）；**1980 **年，企業間為因應商業資訊傳遞的即時性需求，紛紛使用電子資料交換（**Electronic Data Interchange，EDI**）與電子郵件（**Electronic Mail，e-mail**）的技術，將商業文件以標準的電子形式傳遞於企業間，進而提高業務效率並且降低不必要的紙張成本；**1990 **年，全球資訊網的出現，揭開了電子商務時代的序幕。由於網際網路的應用可以明顯地**

提升工作效率、 降低管銷成本、提昇服務品質， 並且具備即時性，因此隱藏著無限商機，故網際網路在人們的期待下於 1992 年開放商業使用之後， 隨即在全球各地掀起一股銳不可當的電子商務潮流。美國政府體認到電子商務對全球未來商務活動的重要性， 於是柯林頓總統於 1997 年 7 月 1 日發表「電子商務框架」（The Framework for Electronic Commerce），提出建立全球電子商務環境之九大議題，以作為電子商務環境發展、資訊科技研發、 國際合作、以及消費者與企業權益保障的行動政策與原則。隨即，得到全球各經濟個體熱烈的討論與回響。

然而，「水可載舟，亦可覆舟」。網際網路為人們帶來的無限商機，但亦可能成為電腦犯罪的溫床。因此，如何建立一個安全及可信賴的網路環境，將是電子商務能否全面普及的關鍵。若是不能建立一個安全可信賴的網路環境，即便電子商務勾勒出美好的未來願景，使用者仍是望而怯步。在此，所謂的「安全及可信賴的網路環境」是確保資訊在網路傳輸過程中不易遭受偽造、非法存取、竄改、竊取或截聽該內容，並且亦能鑑別（Authenticate）交易雙方的身分，以防止事後否認其交易事實。就資訊安全的觀點而言，這樣環境必須能提供下列功能：

1. 完整性（Integrity）：確保網路中所傳輸之資訊與原來的資

訊一致，不會遭到竄改或偽造。

2. 可驗證性（Authentication）：確保網路中之個體（Entity）的身分確實如他所表明的，或由網路所接收的資料確實為該傳送者（Sender）所傳送。

3. 不可否認性（Non-repudiation）：發送端不可否認其所同意傳送出的資料或他所完成的交易行為。

4. 機密性（Confidentiality）：防止非法使用者得知已保護之敏感資料的內容。欲達到上述功能，必須仰賴密碼技術（Cryptography Technology）中的加密技術（Encryption Technology）與數位簽章（Digital Signature）等。除了機密性功能以外，其他功能皆能透過數位簽章的使用來達成，因此接下來將針對數位簽章作介紹。

一、數位簽章與數位簽章機制

現實生活中，為了要向他人證明自己對文件負責，或是證明該文件是由自己所發出的，通常會在文件上加蓋自己的印章或親筆簽名以資證明。例如：傳統的交易行為，一個具有法律效力之商業交易，通常會有書面文件（例如：契約書）並且在其上加以簽名或蓋章，如此才可以確定交易雙方彼此相關的權利與義務。同樣地，在電子商務的環境中，商業交易行為則必須仰賴電子文件與數位簽章來確立其權利與義務。所謂「電子文件」是指與交易行為相關的資訊，其資訊型

態可以為文字、聲音、圖片、影像、符號等之電子形式；而「數位簽章」是指依附於電子文件上，用以辨識及確 認電子文件簽署者（Signer）之身分及電子文件真偽的資訊。由此可知，數位簽章的功能與印章或親筆簽名的功能相似，只是它所針對的文件是電子文件，而非實體文件。

另外，傳統的印章或親筆簽名與數位簽章最大的差異，除了所欲簽署的文件之形式不同外, 印章或親筆簽名與該文件內容是各自獨立且無關的。換言之，針對不同的文件，簽署者使用印章或親筆簽名以產生的簽章不會隨著文件內容不同而有所不同。然而，從數位簽章的產生過程來看, 數位簽章是透過電子文件與簽署者所擁有的秘密資訊， 例如：密鑰（Secret Key）經簽章產生機制（類似數學函數）計算所得的結果。因此，數位簽章與電子文件的內容息息相關，亦即，同一位簽署者所產生的數位簽章，會隨著電子文件內容不同而有所不同。

目前實作上，數位簽章是以密碼學上的公開金鑰密碼系統（Public Key Cryptosystem），又稱「非對稱密碼系統（Asymmetric Cryptosystem）」為基礎來實作，亦即在該系統中，每一位使用者必須自行產生自己所擁有的金鑰對（Key Pair）：一把密鑰與一把公鑰（Public Key）。其中使用者必須秘密地保存自己的密鑰，並且將其公

鑰公佈於網路中。之後，使用者可以利用自己的密鑰對文件進行簽署；而數位簽章的接收者可以利用該簽署者的公鑰來驗證數位簽章的有效性。

一個安全且有效的數位簽章，除了簽署者必須要以正確且有效的方法來對電子文件進行簽署外，其所產生的數位簽章之有效性亦需要一個合適的驗證方法來驗證。數位簽章機制（Digital Signature Mechanism）便是以密碼學（Cryptography）為基礎來定義安全的簽章產生與簽章驗證方法，此機制包括：簽章產生機制（Signature Generation Mechanism）與簽章驗證機制（Signature Verification Mechanism）。「簽章產生機制」是指簽署者產生數位簽章的方法或程序，而此機制可視為一個數學演算法。若簽署者要進行簽署時，他可以將欲簽署的電子文件與自己所擁有的密鑰當作該演算的輸入值，經過該演算法的計算後便能得到電子文件的數位簽章。另一方面，「簽章驗證機制」是指驗證者用來驗證數位簽章之有效性的方法或程序。若是驗證者收到簽署者的電子文件與數位簽章時，他必須使用電子文件、數位簽章以及簽署者的公鑰，並且透過此機制來驗證此數位簽章的有效性。

與數位簽章息息相關的密碼技術為「單向雜湊函數」（One-Way Hash Function），此單向雜湊函數是一種可以將任意長度的輸入值壓

縮成固定長度之輸出值的數學函數或演算法，並且無法從其輸出值去推算其輸入值。在安全性（亦即防止非法者偽造一個合法的數位簽章，以及防止攻擊者從簽章訊息破解出簽署的密鑰）與效率性的考量下，安全的數位簽章機制必須引入單向雜湊函數於該機制中。換言之，在簽章產生機制中，簽署者必須先透過單向雜湊函數將電子文件轉換成固定長度的位元資料，稱之為資料摘要（Data Digest），隨後再使用密鑰簽署該資料摘要以產生數位簽章；同樣地，驗證者亦需先使用此單向雜湊函數，將電子文件轉換成固定長度的資料摘要再進行驗證動作。如圖 2-8 所示。

目前較普遍的數位簽章機制有：RSA、ElGamal 以及 DSA。假設 M 為所欲簽署的電子文件，而 H 為一單向雜湊函數。這三種機制分別詳述如下：

- RSA 數位簽章機制：1978 年，Rives、Shamir 及 Adleman 三位學者利用分解大質數的困難度，提出 RSA 數位簽章機制。目前，VISA、MasterCard、IBM、Microsoft、HP 等公司所協力制定的安全電子交易標準（Secure Electronic Transactions，SET）便是採用 RSA 數位簽章機制。系統設置時，每一位使用者可以先選擇其密鑰：二大質數 (p, q) 以及一整數 d<(p-1)(q-1)，其中 d 與 (p-1)(q-1) 互質；之後再計算出其公鑰： N=p*q 與 e=d-1mod(p-1)(q-1)，

其中 mod 表示模數運算（亦即取餘的運算）。

- ElGamal 數位簽章機制：T. ElGamal 於 1985 年提出 ElGamal 數位簽章機制，而此機制的安全性是建立在解決離散對數問題的困難度上。在使用此機制之前，系統會先公佈一個大質數 p 和模 p 的原根 g_0 之後，每位使用者先任選一個小於 p-1 的整數 x 作為密鑰，並且計算出他的公鑰 $y=g^x \bmod p$。
- DSA 數位簽章機制：此數位簽章機制是由美國國家標準局（National Institute of Standard and Technology，NIST）於 1991 年 8 月提出，其安全性與 ElGamal 數位簽章機制相同，皆建立在解決離散對數問題的困難度上。在使用此機制之前，系統會先選擇一個 512 位元的質數 p 與一個 160 位元的質數 q，其中 p-1 可以被 q 整除。另外，系統再任選一個小於 p-1 的整數 h 並且計算出 $g=h^{(p-1)/q} \bmod p$。最後，公佈系統參數{p, q, g}。使用者一旦得知系統公開參數之後，便可任選一個小於 q-1 的整數 x 作為密鑰，並且計算出他的公鑰 $y=g^x \bmod p$。

二、數位簽章原理

1. 簽章過程及說明如下：

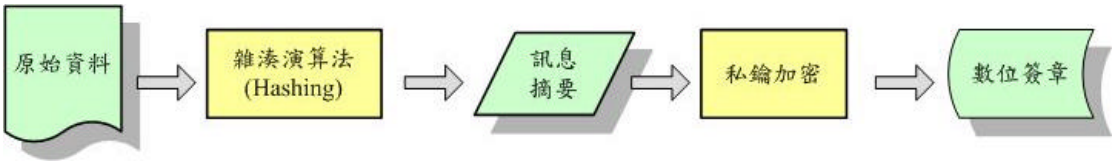

圖 2-7　簽章過程

[1]. **將原始資料利用雜湊演算法**(Hashing)**轉換為訊息摘要, 再利用私密金鑰對訊息摘要進行加密運算即可得到此原始資料之數位簽章。**

[2]. **所使用之雜湊演算法具備「單向不可逆運算」之特性，僅能由原始資料推算出訊息摘要, 而無法由訊息摘要反向推算出原始資料之內容, 因此原始資料與訊息摘要之內容具有關聯性, 且不同之原始資料內容不會運算出相同之訊息摘要，我們可以將訊息摘要視為精簡版之原始資料特徵。**

[3]. **為節省簽章所需運算時間, 因此對較為簡短的訊息摘要進行簽章，而不對原原始資料進行簽章；惟因訊息摘要與原原始資料內容完全相關, 對訊息摘要簽章即相當於對原原始資料簽章。**

[4]. **加密運算是一個相當複雜之運算過程, 由於其破解困難度非常高(以目前電腦速度需數萬年以上)，只要私密金鑰不外洩，他人即無法偽造代表交易資料之數位簽章，因此，數位簽章即可達到傳統印章之身分識別功能。**

2. **驗證簽章過程及說明如下：**

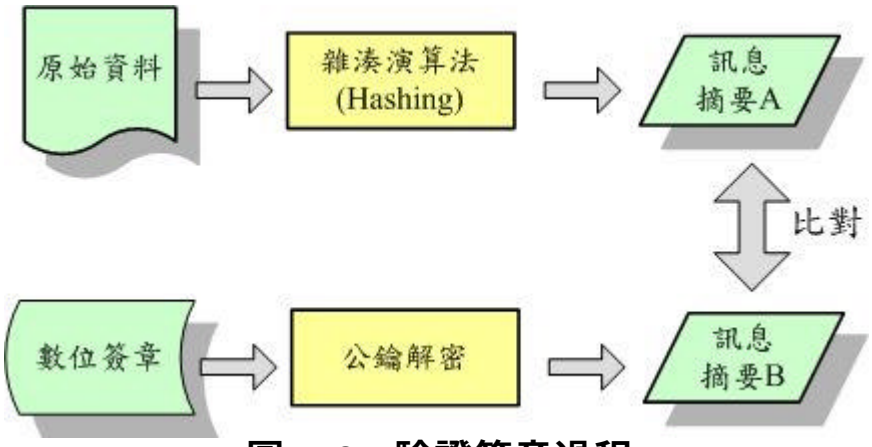

圖 2-8 驗證簽章過程

[1]. **當接收端收到原始資料及數位簽章後, 依其接收之原始資料經雜湊運算產生訊息摘要 A（如上圖）。**

[2]. **利用私密金鑰配對之公開金鑰可將數位簽章以解密運算還原為原來的訊息摘要 B, 比對訊息摘要 A 與訊息摘要 B 作，若兩者相同即表示原始資料或數位簽章正確無誤。**

[3]. **公開金鑰與私密金鑰具有配對關係, 經某私密金鑰簽章之資料，只能由其配對之公開金鑰才能正確完成驗證。**

[4]. **認證機構(網路認證公司)證明公開金鑰之擁有者，並將公開金鑰置於電子憑證中公開，供接收端使用。**

透過上述機制可確認所接收資料的正確性。

三、數位簽章應用

就電子商務環境所需提供的功能而言， 使用數位簽章可以用來達到資訊的完整性、可驗證性與不可否認性等功能， 詳述如下（假

設每位使用者不會將其密鑰洩露給他人）：

1. 完整性：由於數位簽章與電子文件息息相關，所以一旦電子文件被非法變更，則數位簽章將無法通過驗證機謇瘍禈牷 C 例如：使用者 A 欲透過網路下單買股票，A 可以將買單“買進 1000 股”傳送給營業員。然而，如果 A 沒有針對該訊息進行完整性的保護，則非法攻擊者可能會將買單改為”買進 2000 股”，如此將影響 A 的權益。這種情況可以透過數位簽章的使用來防止，亦即 A 除了傳送買單以外，尚需要傳送買單的數位簽章給營業員。 若是營業員發現該數位簽章驗證無效時， 則表示此買單可能已被修改，因此可以要求 A 再次傳送買單。

2. 可驗證性：公開金鑰密碼系統中，每位使用者的公鑰與其密鑰有其唯一的對應關係， 換句話說，只有使用者所擁有的密鑰才能對應到使用者的公鑰， 因此藉由此金鑰對可以達到驗證使用者身分的功能。 數位簽章機制中，簽章產生機制必須使用簽署者的密鑰，簽章驗證機制則要使用簽署者的公鑰，才能驗證該簽章的有效性。 因此，如果驗證者利用公鑰驗證所收到數位簽章為有效時，則表示此簽章與電子文件的確是由具有該公鑰的使用者所簽署。承上例，營業員可將買單、買單的數位簽章以及 A 的公鑰輸入於簽章驗證機制中，若該機制輸出“有效”時，則表示此買單的確是 A 的傳送過來的。

3. **不可否認性：為了能防止傳送端抵口否認曾經發出訊息給接收端，則接收端必須握有傳送端傳送過該訊息的證據。當發生糾紛時，接收端才能夠提出該證據以資證明。承上例，營業員收到** A **的買單與數位簽章之後， 若此數位簽章為有效時，則將此數位簽章與買單儲存起來，以防止** A **事後否認。如果不幸** A **在事後的確否認先前曾經下單給營業員時， 則營業員可以將先前儲存起來的數位簽章與買單，提供給糾紛的仲裁者。此時，仲裁者若驗證此數位簽章為有效時，則他將判營業員勝訴。**

四、數位簽章相關標準法案

1. **數位簽章機制的相關國際標準如下：**

- " Proposed federal information processing standard for digital signature standard (DSS), " Federal Register, Vol. 56, No. 169, 1991, pp. 42980-42982.
- ISO/IEC 14888-1, " Information technology-Security techniques-Digital signatures with appendix-Part 1: General ", International Organization for Standardization, Geneva, Switzerland, 1999
- ISO/IEC 14888-2, " Information technology-Security techniques-Digital signatures with appendix-Part 2: Identity-based mechanisms ", International Organization for Standardization, Geneva, Switzerland, 1999.
- ISO/IEC 14888-3, " Information technology-Security techniques-Digital signatures with appendix-Part 3: Certificate-based mechanisms ", International Organization for Standardization, Geneva, Switzerland, 1999.

2. **單向雜湊函數的相關國際標準如下：**

- ISO/IEC 10118-1, " Information technology-Security techniques-Hash-functions-Part 1: General ", International Organization for Standardization, Geneva, Switzerland, 1994.
- ISO/IEC 10118-2, " Information technology-Security techniques-Hash-functions-Part 1: Hash-functions using an n-bit block cipher algorithm ", International Organization for Standardization, Geneva, Switzerland, 1994.
- ISO/IEC 10118-3, " Information technology-Security techniques-Hash-functions-Part 3: Dedicated hash-functions ", International Organization for Standardization, Geneva, Switzerland, 1994.
- ISO/IEC 10118-4, " Information technology-Security techniques-Hash-functions-Part 4: Hash-functions using an n-bit block cipher algorithm ", International Organization for Standardization, Geneva, Switzerland, 1994.
- National Institute of Standards and Technology, NIST FIPS PUB 180, " Secure hash standard, " U. S. Department of Commerce, 1993.

3. **電子簽章法案：**

建立一個安全及可信賴的網路環境，除了需要有密碼技術的支援外，尚需要相關法律來加以規範，才能使電子商務交易行為具有法律效力。有鑑於此，行政院 NII 小組決議由行政院研究發核委員會會同行政院法規會、法務部、財政部、經濟部、交通部、工研院電通所、資策會科法中心及相關學者專家組成「數位簽章法研擬小組」，負責研擬草案。該小組所研擬之電子簽章法草案於民國八十八年十二月二十三日通過行政院院會審議並且於民國九十一年四月一日通過立法院的審議成為正式的法律。另外，目前電子簽章法已立法的國家

有：美國猶他州（1995）、義大利（ 1997 年）、德國（1997 年）、馬來西亞（1997~）、以及新加坡（1998 年）等，而目前正在立法的國家有：英、法、澳、日等國家。

第四節 SSL 協定

在網際網路日益發達的今日，網路使用者透過網際網路來從事各種便利的活動，例如電子郵件、電子購物、資料傳輸等，在開放性網路架構下，數位化資料傳輸的安全性，成為不可或缺的重要問題；在毫無安全機制下的網路中，使用者可能面臨資料遭到冒名傳送、竄改、偽造、不法擷取等問題。為保護資料於網際網路傳送時具安全性及可信賴性，Netscape Communications 公司首先設計 SSL (Secure Socket Layer) Protocol，其介於應用層(Application Layer)及傳輸層(Transport Layer)之間，並將 SSL Protocol v3.0 之網路草案文件(Internet-draft)公佈於網站，在 1999 年被 IETF(Internet Engineering Task Force)接受後，更名為 TLS (Transport Layer Security) 1.0 版，是為 RFC 2246，並被廣泛應用在電子商務的安全機制。 SSL Protocol v3.0 主要具備以下特性：

- 連線具隱密性，於交握協定時產生秘密金鑰(secret key)，以此秘密金鑰來對傳送之訊息進行加解密。
- 點對點(peer to peer)之間的身分驗證採非對稱式加密法

(Asymmetric cryptographic)。

- **連線具可信賴性，訊息傳送時包含訊息完整性之檢查，使用具金鑰之訊息鑑別碼**(keyed Message Authentication Code, MAC)。

一、SSL 協定架構

SSL **協定架構於開放性系統互連**(Open System Interconnection)，**介於應用層**(Application Layer)**與傳輸層**(Transport Layer)**之間**(圖 2-9)，**主要提供網際網路中相互連結的兩個應用層之間相互通訊時的私密性**(privacy)**與可靠性**(reliability)。SSL **協定是層次式的協定**(layered protocol)，**由兩層子協定所組成，分別為紀錄協定**(SSL Record Protocol) **與交握協定**(SSL Handshake Protocol)。

- **紀錄協定**(SSL Record Protocol)：

  SSL **協定的較低層，主要目的是藉由可靠的傳輸協定**(TCP)，**將各種不同之較高層協定包裝後再傳送。**

- **交握協定**(SSL Handshake Protocol)：

  在兩個應用層之間接收或傳送資料前，提供伺服器與用戶端之間相互認證(authenticate)**之機制，並協議雙方通訊使用之加密演算法**(cryptographic algorithm)**及產生加解密金鑰**(cryptographic key)。

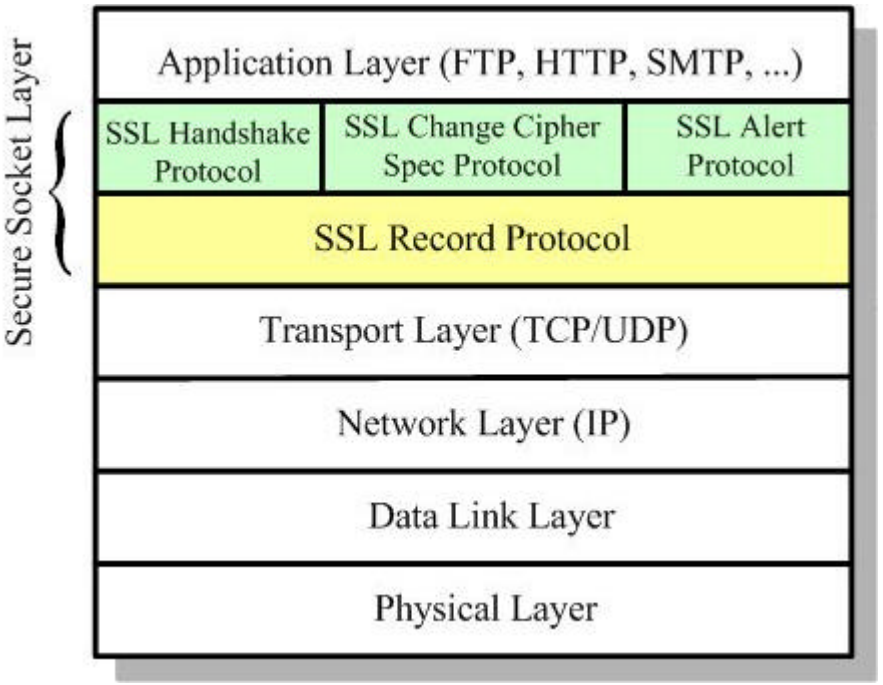

圖 2-9　SSL **安全協定**

另外，在 SSL **協定中，**SSL **的狀態依其參數可分為兩種，分別為交談狀態**(session state)**及連線狀態**(connection state)**，此兩個狀態的定義如下：**

1. **交談狀態**(session state)

交談是指用戶端與伺服器之間的聯合(association)**，交談於交握協定時建立，並定義一組安全參數**(cryptographic security parameter)**，其中包含此交談使用之加解密演算法與交談金鑰**(session key)**，在多個連線中之任一連線皆可分享使用該參數。建立一個交談所需要的參數如下：**

■ **交談識別碼**(session identifier)**：伺服器端會任意挑選一個位元組序列用以辨識一個正在運作**(active)**或可重新開始**(resumable)**的交談**

狀態。

- **節點認證**(peer certificate)：**此參數為** X509 v3 **之憑證，交談狀態中的此資料欄位可以為空白。**
- **壓縮方法**(compression method)：**在資料區塊加密前使用的壓縮演算法。**
- **密文規格**(cipher spec)：**定義密碼規格，包含資料的加密演算法、訊息驗證碼演算法。此參數並定義了一些密碼相關的屬性，例如雜湊值的大小。**
- **主要秘密**(master secret)：**一個** 48 **位元組值，由伺服器端及用戶端所共同使用。**
- **是否可回復**(is resumable)：**旗標，用來表示此交談狀態可否開始一個新的連線。**

2. **連線狀態**(connection state)

一個連線就是一次傳輸程序(根據 OSI **層級模型的定義)，它提供了一個適當的傳輸方式。對於** SSL **而言，這種連線是點對點**(peer to peer)**的關係，並且此連線是暫時的**(transient)**。每一連線對應一個交談**(session)**。建立連線所需要的相關參數如下：**

- **伺服器與客戶端亂數**(server and client random)：**在每次連線建立時，由伺服器端與用戶端選擇之位元序列亂數。**

- **由伺服器寫入** MAC **秘密**(server write MAC secret)**：伺服器端針對送出的資料來產生** MAC **時所需的秘密金鑰。**
- **由用戶端寫入** MAC **秘密**(client write MAC secret)**：用戶端針對送出的資料來產生** MAC **時所需的秘密金鑰。**
- **由伺服器寫入金鑰**(server write key)**：伺服器端使用於加密資料之金鑰，同時也是用戶端用來解密之金鑰。**
- **由用戶端寫入金鑰**(client write key)**：用戶端使用於加密資料之金鑰，同時也是伺服器端用來解密之金鑰。**
- **起始向量**(initialization vectors)**：當我們使用** CBC **模式的區塊加密法時，每一把金鑰都會有一個起始向量**(IV)**。此欄位的初始值是由** SSL **的交握協定來設定。之後，每個紀錄所產生的密文區塊就會成為下個紀錄的起始向量。**
- **序號**(sequence numbers)**：每個節點會針對不同的連線所發出/接收的訊息來維護一組獨立的序號。當雙方收到** change cipher spec **訊息之後，此序號值將被設定成為零;序號不得超過** 2^{64}-1**。**

二、交握協定層(Handshake Protocol)

交握協定在整個 SSL **中是最複雜的一部份，主要目的在讓用戶端與伺服器端之間能相互認證**(authentication)**，並協議彼此加密及產生訊息鑑別碼的演算法以產生用來保護欲傳送之資料安全的金鑰。在**

進行任何 SSL 的連結之前，一定要先經過交握協定協議這些安全參數後，才能開始傳輸資料。建立一個新的安全通道連線的交握協定的流程(圖 2-10)分為下列四個主要步驟：

1. **建立安全環境** (Establish Security Capabilities)

此一步驟主要為用戶端與伺服器端進行初始化的邏輯連結，協議彼此使用的安全環境參數。首先由用戶端送出 Client_hello 的訊息給伺服器端，伺服器端收到後送回 Server_hello 的訊息，進行初始參數值的協議。其中這兩個訊息包含下列欲進行協議的初始參數：

- Version：所使用 SSL 的最高的版本。
- Random：亂數，包含 32 位元的時間戳記(timesamp)及 28 位元的亂數產生器產生之亂數。
- Session ID：欲建立交談之識別碼。用戶端送出時，此值若為 0，則為欲建立一個新的交談；非零值代表欲接續之前的交談。伺服器端送出則為可以建立之交談識別碼。
- CipherSuite：加密套件列表。
- Compression Method：壓縮方式列表。

2. **伺服器端認證及金鑰交換** (Server Authentication and Key Exchange)

在此步驟中，首先由伺服器端傳送 certificate 訊息，並將伺服器端之憑證送出，此憑證為符合 X.509 規格的單一憑證或憑證鏈(如果

伺服器端是使用匿名的 Diffie-Hellman 方式，則不需要傳送此訊息)。接著送出 server_key_exchange 的訊息，但如果在以下兩種情況下，就可以不需送出這個訊息：(1)伺服器端已經利用固定 Diffie-Hellman 的各項參數來傳送憑證，(2)即將進行 RSA 金鑰交換。以下為 server_key_exchange 訊息內容：

- RSA：伺服器端產生一組臨時的 RSA 金鑰對，並在 server_key_exchange 訊息中放入公開金鑰參數(指數跟模數)，目的是稍後讓用戶端用以加密前置主秘密(pre-master secret)用。
- Anonymouse Diffie-Hellman：訊息內容為伺服器端的公開 Diffie-Hellman 參數(質數與原根)，再加上伺服器端的 Diffie-Hellman 公開金鑰所組成。
- Fixed Diffie-Hellman：訊息內容是由三個提供給匿名的 Diffie-Hellman 所使用的參數, 以及這三個參數的簽章所組成。
- Fortezza：送出用戶端的 Fortezza 參數。

接著，如果伺服器端是一個非匿名的伺服器(不是使用匿名 Diffie_Hellman 的伺服器)可能需要從用戶端獲得一個憑證，便會送出 certificate_request 訊息，此訊息包含兩個參數：憑證類型、憑證中心，憑證類型參數指定了所使用的公開金鑰演算法與憑證的用途, 憑證中

心參數則是一個列表，用來記錄可接受的不同憑證中心。最後，伺服器端收到 certificate 訊息，再送出 server_hello_done 的訊息，表示伺服器端的這個階段已完成。

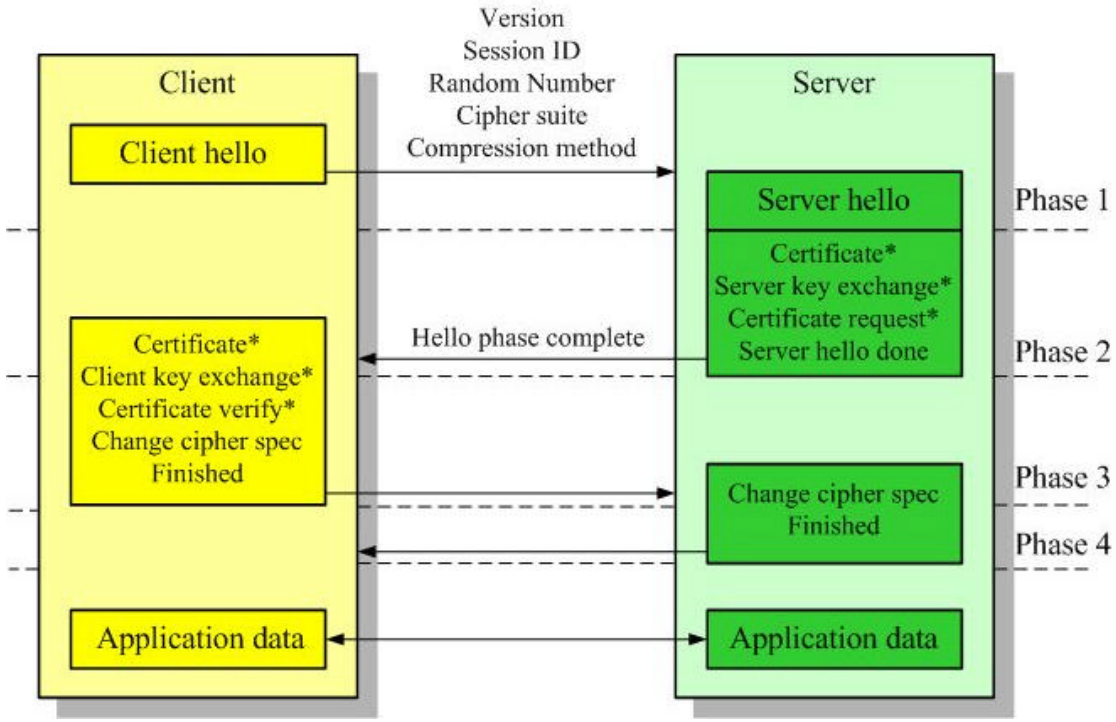

圖 2-10 SSL 交握協定流程圖

3. **用戶端認證及金鑰交換** (Client Authentication and Key Exchange)

在此步驟中，若用戶端接收到伺服器端傳送之 certificate_request **訊息，則傳送** certificate **訊息，並將用戶端的憑證送出，同樣的，此憑證為符合** X.509 **規格的單一憑證或憑證鏈。接著送出** client_key_exchange **的訊息，這個訊息是在此步驟用戶端必定要傳送的訊息，訊息內容則依金鑰交換的形式而定：**

- RSA：**用戶端產生一個** 48 **位元組的前置主秘密**(pre-master

secret)，並且以伺服器端的公開金鑰(由伺服器端的憑證中取出)來加密，或是利用 server key exchange 訊息中臨時的 RSA 金鑰對來加密。

- Anonymouse Diffie-Hellman：送出用戶端的公開 Diffie-Hellman 參數。
- Fixed Diffie-Hellman：用戶端的公開 Diffie-Hellman 參數被包在送出的訊息中，所以這個訊息並未含有任何的資訊。
- Fortezza：送出用戶端的 Fortezza 參數。

最後，用戶端會送出一個 certificate_verfity 訊息，讓伺服器端藉此來驗證用戶端的憑證。這個訊息必須在用戶端已經送出任何一個具有簽章的憑證的情況下才會送出。這個訊息會根據之前所送出的訊息來簽署一個雜湊值。

```
CertificateVerfi.signature.md5_hash=
MD5(master_secret||pad_2||MD5(handshake_messages||
        master_secret||pad_1))
    Certificate.signature.sha_hash=
SHA(master_secret||pad_2||SHA(handshake_messages||
        master_secret||pad_1))
```

其中的 pad_1 跟 pad_2 是先前 MAC 中所定義的數值。Handshake_messages 泛指自 client_hello 訊息之後所傳送的所有交握訊息，但不包含 client_hello 這個訊息。master 則是算出的數值，在之

後會討論。如果使用者的私密金鑰採用 DSS 的話，那麼就用來加密 SHA-1 的雜湊值。如果使用者的私密金鑰採用 RSA 的話，那麼就用來加密 MD5 與 SHA-1 雜湊值得到合併值。此一目的在驗證用戶端憑證中的私密金鑰的所有權，就算有人誤用或盜用用戶端的憑證資訊，也無法傳送這個訊息。

4. **完成**(finished)

此步驟在建立一個安全連結。用戶端送出 change_cipher_spec 的訊息後，以前面步驟協議完成的加密演算法、金鑰及秘密資訊來對 finished 訊息做訊息封裝的動作，傳送至伺服器端，由伺服器端驗證此訊息，如果驗證成功，表示前述的金鑰交換及認證過程正確無誤。驗證無誤後，由伺服器端同樣送出 finished 訊息給用戶端做驗證。finished 訊息內容包含兩個雜湊值：

```
MD5(master_secret||pad_2||
MD5(handshake_messages||Sender||master_secret||pad_1))
SHA(master_secret||pad_2||
SHA(handshake_messages||Sender||master_secret||pad_1))
```

在交握協定的四個步驟中，以雙方協議的安全資訊，來產生 pre_master_secret(如採取 RSA 做金鑰交換，此值為用戶端產生；如採取 Diffie-Hellman 做金鑰交換，此值為用戶端與伺服器端共同產生)，再由用戶端與伺服器端雙方各別計算出主秘密(master_secret)，再藉由

計算出之主秘密導出此 SSL 連結所用來保護資料安全的金鑰。

```
master_secret=MD5(pre_master_secret||SHA('A'||pre_master_secret||
                    ClientHello.random||ServerHello.random))||
              MD5(pre_master_secret||SHA('BB'||pre_master_secret||
                    ClientHello.random||ServerHello.random))||
              MD5(pre_master_secret||SHA('CCC'||pre_master_secret||
                    ClientHello.random||ServerHello.random))
```

此處的 ClientHello.random 及 ServerHello.random 是在一開始雙方的 hello 訊息交換中所採用的臨時亂數。主秘密產生後，雙方便可利用主秘密來產生交談金鑰(session key)。

```
key_block=MD5(master_secret||SHA('A'||master_secret||
                    ServerHello.random||ClientHello.random))||
              MD5(master_secret||SHA('BB'||master_secret||
                    ServerHello.random||ClientHello.random))||
              MD5(master_secret||SHA('CCC'||master_secret||
                    ServerHello.random||ClientHello.random))||…
```

這個步驟一直持續到產生足夠的輸出為止，其結果便是一個虛擬的亂數函數。我們可以將主秘密視為這個虛擬的亂數函數的種子(seed)，而用戶端與伺服器端的亂數可被視為增加破解困難度的醃製劑(salt)。

上述之步驟，便是通訊雙方在 SSL 協定下建立安全連線的交握協定，如果用戶端欲跟伺服器端繼續先前的連結(先前之連線可能因時間過長或網路壅塞等原因造成斷線)，則此時進行要求接續交談的

交握協定跟上述之步驟有所差異(圖 2-11)。用戶端欲接續 SSL 交談，在交握協定的步驟一中，由用戶端送出 Client_hello 的訊息給伺服器端，所傳送的訊息格式與前面所介紹的相同；不同的是，Session ID 的內容為欲接續之交談識別碼。伺服器端收到後送回 Server_hello 的訊息，若同意用戶端接續交談，則回送可以建立連線之交談識別碼，若不同意接續交談，則送出連線失敗的警告訊息。

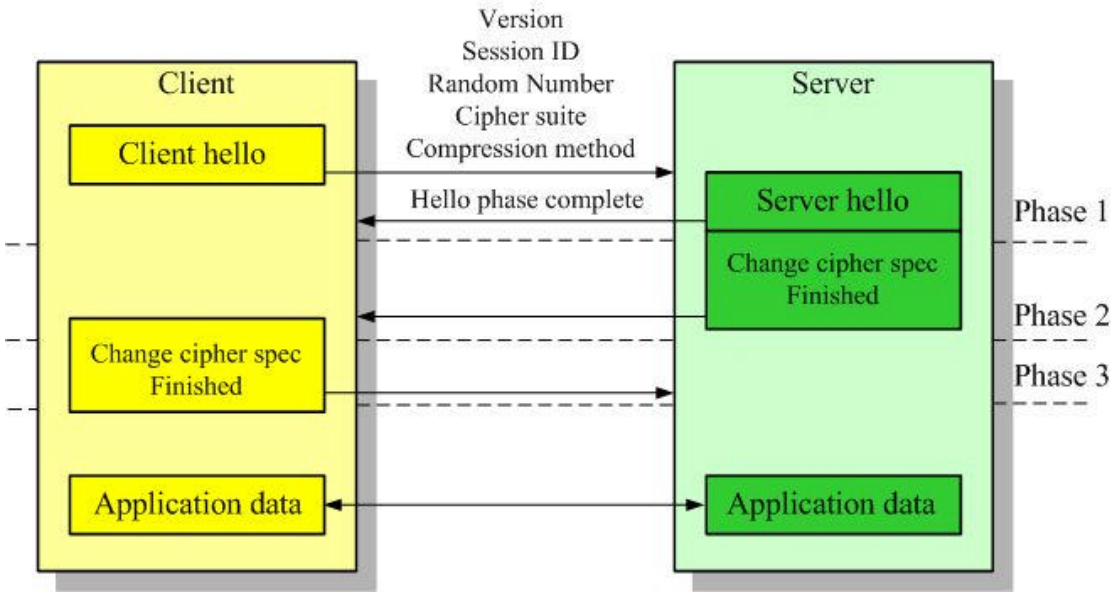

圖 2-11 SSL 交握協定流程圖(接續交談)

三、紀錄協定(Record Protocol)

紀錄協定主要負責的工作，是將欲傳輸之不固定長度的資料訊息，經過一連串的分割/組合(Fragmentation/Assembling)、壓縮/解壓縮(Compression/ Decompression)、加/解密(Encryption/Decryption)等處理，再傳送至上層(或下層)。當使用者欲傳送資料時，資料經由應用

層傳送至紀錄協定層，經過分割(fragmented)、壓縮(compressed)、加入訊息鑑別碼(MAC)、加密(encrypted)之後，加上 SSL 紀錄標頭(SSL record header)，再傳送至傳輸層(圖 2-12)；

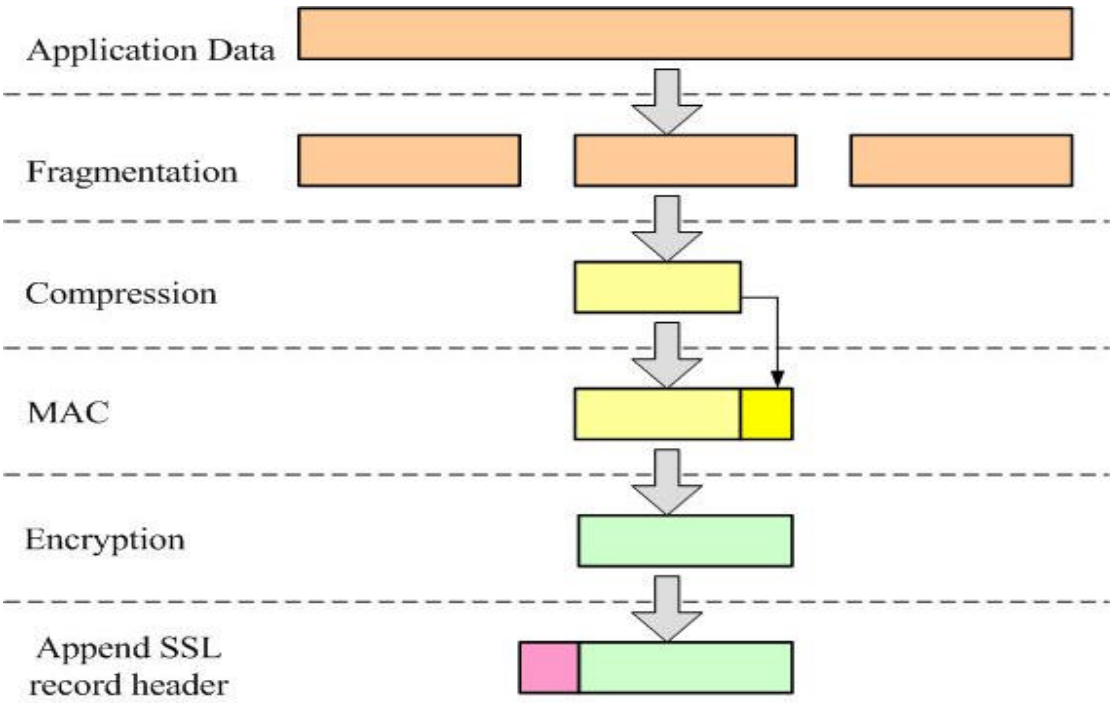

圖 2-12 SSL 紀錄協定之資料封裝步驟

反之，當使用者接收資料時，資料經由傳輸層送至紀錄協定層，經過解密(decrypted)、驗證訊息鑑別碼(verified)、解壓縮(decompressed)、組合(reassembled)後，再將資料送至應用層。紀錄協定針對 SSL 連結提供兩大服務：

- 保密性(Confidentiality)：在 SSL 交握協定中，由連結的兩端協議出共享的秘密金鑰；以此金鑰來對傳送資料做加密，以達到保密性。

■ **訊息完整性**(Message Integrity)：**在** SSL **交握協定中，由連結的兩端協議出共享的秘密金鑰；以此金鑰來對傳送資料加入訊息鑑別碼**(MAC)，**以達到訊息完整性。**

上述 SSL **紀錄協定中，對於欲傳送/接收的資料會經過一連串的封裝處理步驟，針對傳送之資料的處理過程如下(接收之資料則為反向動作)：**

1. **分割** (Fragmentation)

將應用資料分割成為 SSLPlaintext records，**其區塊大小不可大於** 2^{14} **位元組**(16,384 bytes)。**分割後區塊內容包括上層資料型態、**SSL **版本、分割後明文長度及分割後明文。**

2. **壓縮** (Compression)

將 SSLPlaintext **壓縮成為** SSLCompressed records。**壓縮後區塊內容包括上層資料型態、**SSL **版本、壓縮後資料長度及壓縮後資料，壓縮後長度不得超過未壓縮之區塊** 1024 **位元組，也就是壓縮後之區塊長度不得超過** 2^{14}+1024 **位元組。**

3. **訊息鑑別碼**(Message Authentication Code, MAC)

為了計算已壓縮資料的訊息鑑別碼，必須使用一把共享的秘密金鑰，其計算過程如下：

hash(MAC_write_secret||pad_2||
hash(MAC_write_secret ||pad_1||seq_num||
SSLCompressed.type||SSLCompressed.length||
SSLCompressed.fragment)

4. **加密** (Encryption)

加密演算法包含串流加密法(stream cipher)**、區塊加密法**(block cipher)**，其中區塊加密法亦提供** CBC **操作模式。加密後區塊內容包括上層資料型態、**SSL **版本、加密後密文長度及加密後密文，加密後之區塊長度不得超過** 2^{14}+2048 **位元組。**

SSL **記錄區塊經過上述步驟的封裝後，會在資料前端加上標頭**(**圖** 2-13)**，再傳送給下一層傳輸協定處理；標頭其欄位如下：**

- **內文類型**(content type)**：** 8 **位元值，指出用來處理被包裝區塊的較高層協定。**
- **主要版本**(major version)**：**8 **位元值，指出所使用的** SSL **的主要版本；對** SSL v3 **而言，其值為** 3**。**
- **次要版本**(minor version)**：**8 **位元值，指出所使用的** SSL **的次要版本；對** SSL v3 **而言，其值為** 0**。**
- **壓縮後長度**(compressed length)**：**16 **位元值，以位元組為單位的本文區塊的長度，或是經過壓縮的區塊長度。最大值為** 2^{14}+2048**。**

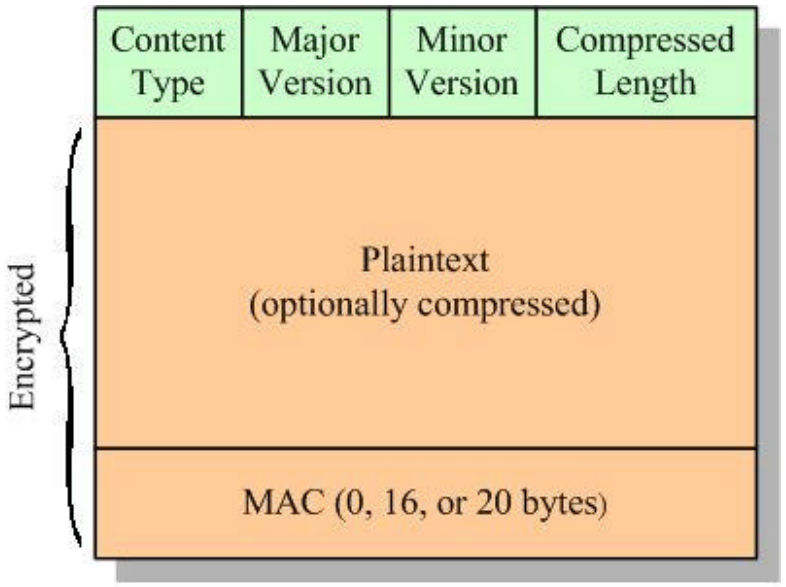

圖 2-13　SSL **紀錄協定之資料格式**

四、變更加密規格協定(Change Cipher Spec Protocol)

在 SSL **的紀錄協定層上，有三個特別的協定，分別為變更加密規格協定**(Change Cipher Spec Protocol)**、警告協定**(Alert Protocol)**與交握協定**(Handshake Protocol)**。**

變更加密規格協定主要在提供 SSL **紀錄協定使用，內容包含** 1 **個位元組的訊息，訊息值為** 1**，主要在提供** SSL **紀錄協定在判別同一個連結中是否變更加密套件**(cipher suite)**。**

五、警告協定(Alert Protocol)

警告協定是用來傳遞與 SSL **相關的警告訊息到另一連結的節點。此協定所傳送的訊息也經過壓縮、加密。警告協定內的每個訊息都由兩個位元所組成，第一個位元組代表警訊的類別，分為以下兩種：**

- **關閉警告** (Closure alerts)
- **錯誤警告** (Error alerts)

警告協定主要負責下列工作：

- **當錯誤警告被偵測到時，偵測端將傳送訊息通知另一端。**
- **若傳送或接收到的訊息是** fatal alert message**，雙方會立即關閉此連線。**

若連線失敗(關閉)，發出關閉警訊，並要求伺服器跟客戶端都需要把交談識別碼(session id)**、金鑰**(keys)**及相關的秘密資訊統統移除。**

第參章 系統架構與流程說明

第一節 系統架構說明

為呈現此快速且安全的電子病歷分享模式，此一系統架構，可分為六個主要的組成單元(圖 3-1)：

- **醫療憑證管理中心**(Health Certificate Authority, HCA)：**架設於衛生署，負責各醫療院所、醫事人員卡和健保** IC **卡的憑證驗證管理工作以及定期更新憑證驗證管理清單。**
- **電子病歷索引中心**(Electrical Health Record Index Center, EHRIC)：**架設於衛生署，負責內容包含：**
  i. **接收各醫療院所上傳的電子病歷索引及其簽章**
  ii. **接受各醫療院所電子病歷查詢的要求, 並回傳電子病歷索引及簽章**
- **醫療資訊系統**(Health Information System, HIS)：**架設於各醫療院所內，負責定時提供電子病歷資料給電子病歷資料庫。**
- **電子病歷資料庫伺服器**(Electrical Health Record Database Server, EHRDB)：**架設於各醫療院所內，負責內容包含：**
  i. **接受醫療資訊系統定期上傳** HL7 **資料，並轉換成** XML **格式**
  ii. **立即或定時產生代表此份文件的病歷索引資料及電子簽**

章並傳送至電子病歷索引中心

iii. 結合 HL7 訊息處理機制及 HL7 格式轉換機制，接受病歷資料的查詢要求及回傳完整的電子病歷資料

iv. 電子病歷管理

- 電子病歷查詢應用伺服器(Application Server , AS)：架設於各醫療院所內，負責內容包含：

  i. 提供電子病歷查詢介面

  ii. 提供電子病歷顯示畫面

  iii. 使用者權限控管，與 IC 卡之整合

  iv. 結合 HL7 訊息處理機制及 HL7 格式轉換機制，發出病歷資料查詢需求及顯示回傳之 HL7 格式的完整電子病歷資料

- 病歷查詢端(End Entity)：裝有支援 SSL 連線服務的瀏覽器的一般 PC 並連結可插入兩張卡(包括健保 IC 卡《National Health Insurance IC Card, NHI IC Card》和醫事人員卡《Hospital Personnel Card, HPC》)由衛生署核發的讀卡機。

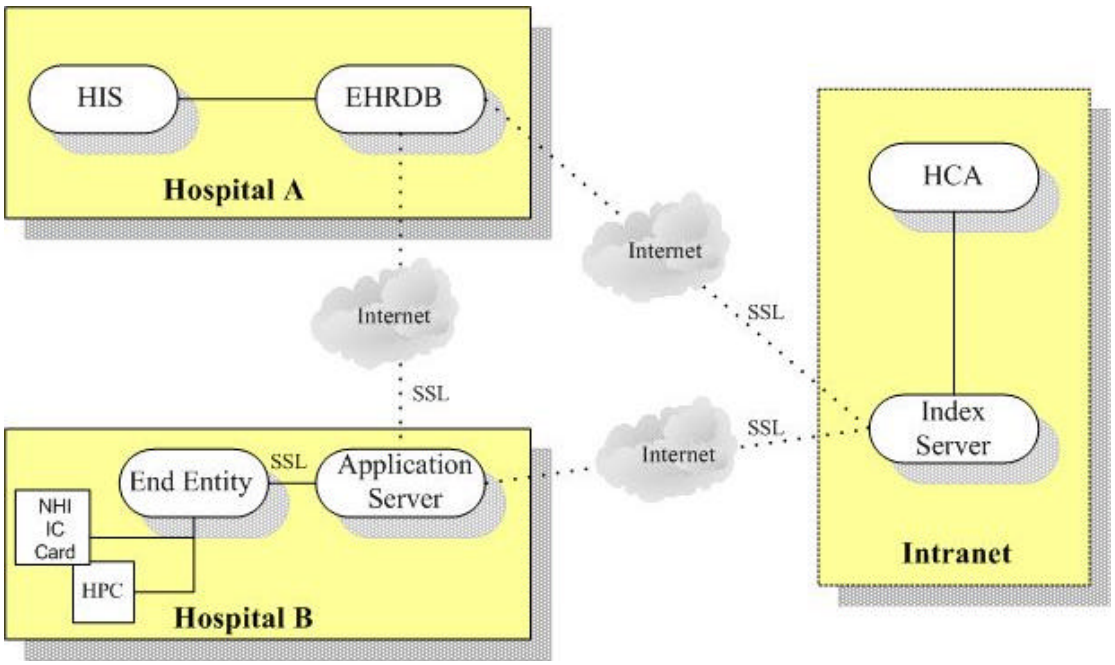

圖 3-1　系統架構圖

第二節 系統流程說明

根據第一節的系統架構，整個系統流程將分為三個階段：第一階段為電子病歷登錄階段，第二階段為電子病歷搜尋階段，第三階段為電子病歷查閱階段。

- **電子病歷登錄階段**(EHR Registration Phase)**：此階段主要在醫療院所將產生的電子病歷索引及電子病歷簽章上傳至電子病歷索引中心存證**(圖 3-2)**。**

R.1. **醫療院所批次地由** HIS **抓出符合** HL7 **標準格式的電子病歷後依事先定義好的** XML **標籤轉換成** XML **格式的電子病歷。**

R.2. **轉換後的電子病歷會先存放在電子病歷資料庫的暫存資料表，電子病歷資料庫在偵測到有新病歷資料存入時，便將病歷資料**

抓出以醫療機構的私密金鑰加以簽章，接著將電子病歷資料與簽章一同存入另一個資料表，如此才算完成電子病歷儲存的工作。

R.3. 將資料庫常用的檢索項目，如看診日期、科別、醫院代碼等，集合成電子病歷索引並連同電子病歷簽章經由 SSL 的保護，透過 HTTPS 協定上傳至電子病歷索引中心登錄到資料庫內。

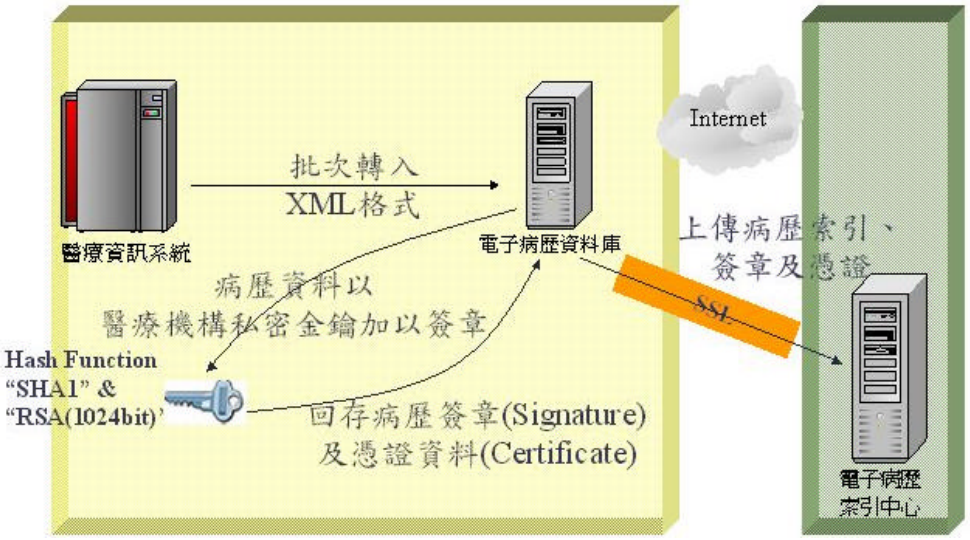

圖 3-2　電子病歷登錄階段

■ 電子病歷搜尋階段(EHR Searching Phase)：此階段主要在使用者透過瀏覽器連結至電子病歷查詢系統網站提供的查詢界面，送出查詢條件至電子病歷索引中心(圖 3-3)。

S.1. 使用者打開瀏覽器連結至電子病歷查詢系統網站，待使用者輸入病歷查詢條件並按下查詢按鈕後，網頁內嵌的 Applet 會偵測是否醫事人員卡和健保 IC 卡都插在讀卡機上。

S.2. 若偵測到沒有插卡或少插一張卡，Applet 會顯示錯誤訊息並不會為使用者傳送查詢條件。

S.3. 若都有插卡，Applet 會讀取健保 IC 卡裡的資料並顯示在網頁上。

S.4. 此時，Applet 將搜尋條件傳給健保 IC 卡做完簽章後附上憑證，接著再將資料傳給醫事人員卡做簽章並附上憑證，最後經由 SSL 保護送至同一院內的應用伺服器。

S.5. 應用伺服器收到後，以其伺服器的私密金鑰對查詢條件資料再加以簽章後，才轉送至電子病歷索引中心。

S.6. 電子病歷索引中心收到應用伺服器傳來的資料後，先解開伺服器的簽章和原資料做 hash 再做比對。確定資料來源是可信任之後，先從憑證管理中心下載新的憑證廢止清單，利用這清單先檢查醫事人員卡和健保 IC 卡的憑證是否過期，再利用醫事人員卡和健保 IC 卡的憑證裡的公鑰驗證由這兩張卡裡的私密金鑰做的簽章是否有效。

S.7. 若檢驗有誤，則回傳驗證錯誤訊息給應用伺服器，最後顯示錯誤訊息至使用者的瀏覽器上。

S.8. 若檢驗無誤，電子病歷索引中心根據查詢條件從資料庫找到所有相對的病歷索引和簽章並組成一個 XML，將此 XML 回傳給應用伺服器。

S.9. **應用伺服器收到後將** XML **格式的電子病歷索引清單轉換成** HTML **的格式並回傳至使用者的電腦顯示在瀏覽器上。**

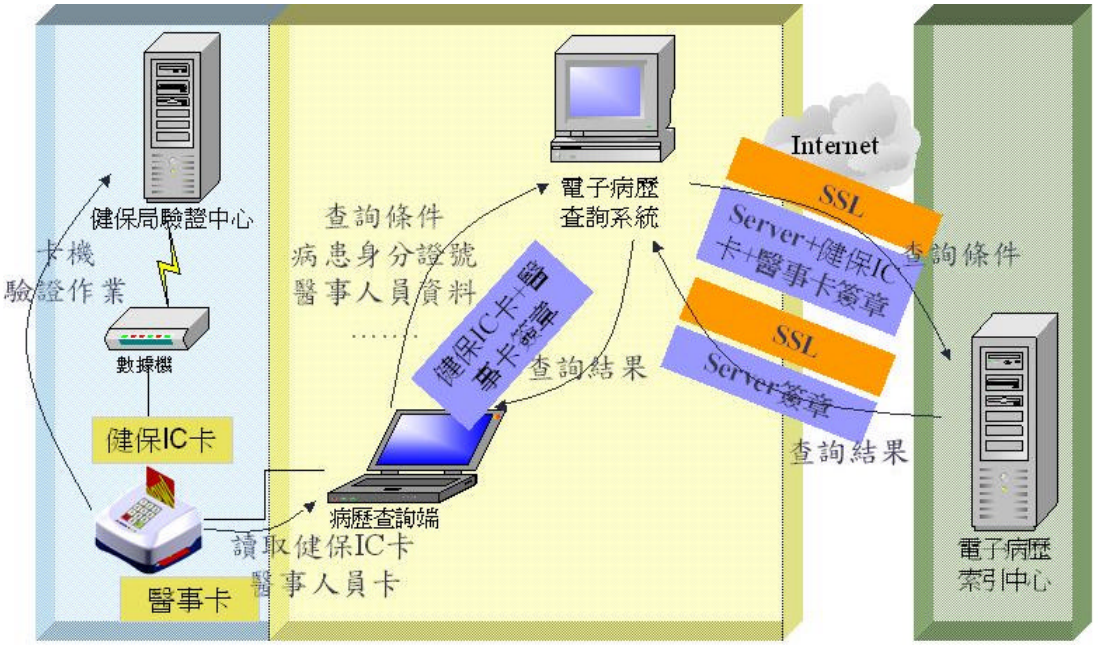

圖 3-3 電子病歷搜尋階段

■ **電子病歷查閱階段**(EHR Consultation Phase)**：此階段主要在當瀏覽器顯示出電子病歷索引清單時，使用者可點選任一筆資料查看該病歷索引的完整病歷資料**(圖 3-4)。

C.1. **當瀏覽器列出電子病歷索引清單後，待使用者點選某一筆病歷索引，網頁內嵌的** Applet **會偵測醫事人員卡和健保** IC **卡是否仍然插在讀卡機上。**

C.2. **若偵測到沒有插卡或少插一張卡，**Applet **會顯示錯誤訊息並不會為使用者傳送查閱資料。**

C.3. **若都有插卡，**Applet **會將此筆電子病歷索引、電子簽章及擁有**

該病歷的醫院 IP 等資訊，經由 SSL 的保護，透過 HTTPS 協定送往應用伺服器。

C.4. 應用伺服器根據查閱資料裡被指定的醫院 IP 經由 SSL 的保護轉送該筆電子病歷索引的查閱要求至該醫院，其中，傳送的格式是遵循 HL7/XML 標準的查詢訊息格式，代碼為 QRY^T12，T12 表示這個查詢要求屬於事件 12，而 QRY 表示傳送的是查詢要求。

C.5. 該醫院先根據收到的病歷索引從電子病歷資料庫裡找到相對的簽章，接著和收到的電子簽章做比對。

C.6. 若檢驗有誤，則回傳驗證錯誤訊息給應用伺服器，最後顯示錯誤訊息至使用者的瀏覽器上。

C.7. 若比對無誤，則回傳資料庫裡 XML 格式的完整電子病歷及簽章給應用伺服器，其中，傳送的格式同樣遵守 HL7/XML 標準的格式，代碼為 DOC^T12，T12 表示這個查詢回應屬於事件 12，而 DOC 表示傳送的是資料檔案。

C.8. 應用伺服器收到後計算這完整電子病歷的摘要(digest)，然後利用電子病歷索引中心的公鑰解開電子病歷索引中心傳來的簽章成另一 digest，將兩者做比對。

C.9. 若比對無誤，則將 XML 格式的電子病歷轉換成 HTML 網頁。

C.10. 將 HTML 回傳至使用者端並顯示在瀏覽器上。

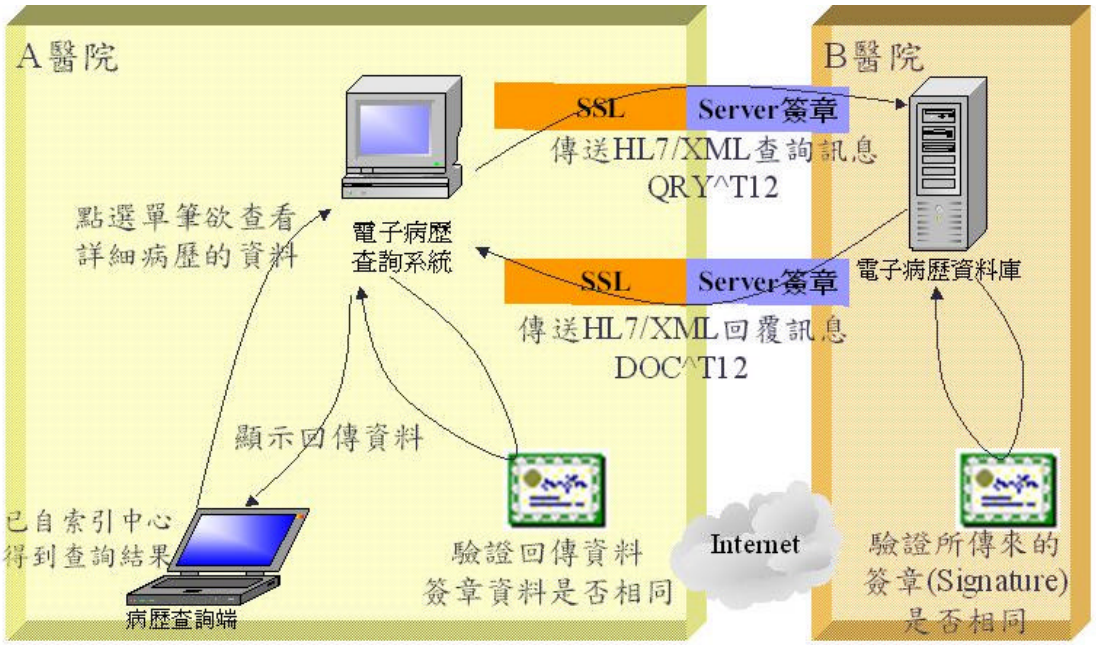

圖 3-4　電子病歷查閱階段

第肆章 系統安全模組實作與應用

第一節 Java Security API和 JSSE 套件研究

一、 Java Security API研究

Java Secuirty API是 Java 新的核心 API，它首次出現於 JDK 1.1 版本裡，所有 java.security 類別跟子類別都屬於 Java Secuirty API。Java Security API 共包括下列三個 API：

- **數位簽章 API：**

  此 API 負責提供產生公開金鑰和私密金鑰對以及簽章和驗證任何數位資料的函式和物件。數位簽章 API 實作的演算法包括 SHA1withDSA、MD5withRSA 和 SHA1withRSA 等。

- **訊息摘要 API：**

  此 API 負責提供建立及驗證訊息摘要的函式和物件。訊息摘要 API 實作的演算法包括 MD5 和 SHA-1 等。

- **金鑰管理 API：**

  此 API 負責提供管理對稱式金鑰、非對稱式金鑰以及憑證的函式及物件。物件包括 KeyStore、KeyManagerFactory 等。

二、JSSE 套件研究

1.JSSE 套件

在 Java 中有許多與網路安全相關套件，如：Java Cryptography Extension(JCE)、Java Secure Socket Extension(JSSE)及 Java Authentication and Authorization Service(JAAS)，另外還有新發展出套件 Java GSS-API 及 Java Certification Path API。其中，最符合實作本論文之系統所需的 SSL 安全連線機制非 JSSE 莫屬。 JSSE 可以讓程式開發人員發展出一個於伺服端與客戶端之間溝通資料安全的架構, 完全由 Java 開發完成, 它的功能包括實作 SSL 及 TLS 傳輸協定, 含伺服器與用戶的鑑別、資料加密及訊息的完整性等。

JSSE 在 J2SDK 1.2 版及 1.3 版裡還不是內建的 API，假使程式中想使用 JSSE 套件得另外去 Sun 的網站下載這份套件，再將它包括進 J2SDK 裡才可以使用, 然而在 J2SDK v1.4 版本裡就將 JSSE 內建進去而不需另外下載及設定就可以使用，也就是說 JSSE 已經被視為基本的 API 了。

現在我們將為您介紹在 J2SDK 1.4 版本中 JSSE 的 API，包括 javax.net、javax.net.ssl 和 javax.security.cert，其功能的特色及優點如下：

- 它是完全由 Java 開發完成。
- 支援 SSL v2.0 及 v3.0，TLS v1.0。
- 提供能夠發展出安全通道(secure channels)的類別，如：

SSLSocket 及 SSLServerSocket。

- 提供伺服端與客戶端的鑑別(authentication)，如：SSL 交握(Handshake)協定。
- 支援私密金鑰及認證機構(Certificate Authority)。
- 支援 HTTPS 協定：比如可以在 HTTPS 協定下取得 HTML 文件。
- 提供多種加密演算法，包含：

加密演算法種類	Key Length(bits)
RSA public key	2048 (authentication), 2048 (key agreement)
RC4	128
DES	64 (56 effective)
Triple DES	192 (112 effective)
Diffie-Hellman public key	1024
DSA public key	2048

2.SunJSSE Provider

在 J2SDK 1.4 版本裡 JSSE 預設的 Provider 是「SunJSSE」。Provider 是指提供某種演算法的廠商，比如 A 公司遵循 JDK 規格，提供了 DSA 演算法，那麼 A 公司就是一個 Provider。Provider 提供的演算法函式庫為"engine algorithms", 簡稱 engines。而 JSSE 中所提供的 engines 類別有 SSLContext、KeyManagerFactory 及 TrustManagerFactory。除此之外，尚有 Java Security 標準的 engines 類別。下面將依不同的類別所提供的演算法及協定分類：

Engines 類別名稱	演算法/協定
KeyFactory	RSA
KeyPairGenerator	RSA
KeyStore	PKCS12
Signature	MD2withRSA, MD5withRSA, SHA1withRSA
KeyManagerFactory	SunX509
TrustManagerFactoy	SunX509
SSLContext	SSL, SSLv3, TLS, TLSv1

第二節 系統安全模組設計

在本論文之系統裡，我負責撰寫安全模組的部分，包括電子病歷簽章驗證程式模組和 SSL 客戶端連線程式模組。這兩個模組皆能以 Java Security API 和 JSSE 函式庫開發。程式模組流程將在以下說明。

一、電子病歷簽章驗證程式模組

1. 電子病歷簽章程式

這個程式分別提供 createPrivateKey( )和 sign( )共兩個函式給其他函式使用。函式說明如下：

- *createPrivateKey*

  電子病歷要簽章前，必需先將私密金鑰物丟進此函式轉換成 PrivateKey 物件，接著才能對電子病歷進行簽章的動作。設計此函式流程(圖 4-1)，輸入字串為私密金鑰在系統裡的路徑，輸出物件為 PrivateKey 物件。

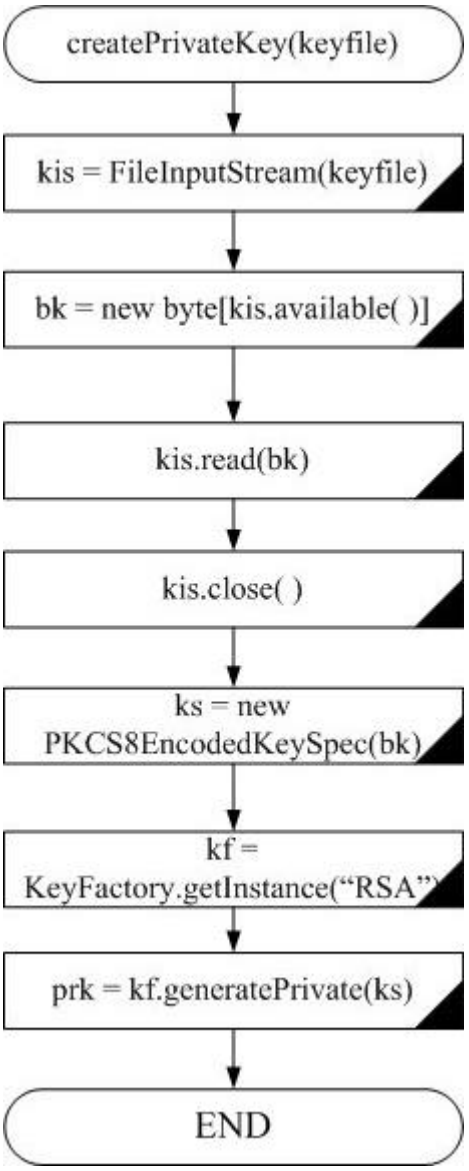

圖 4-1　createPrivateKey( )**函式流程圖**

■ *sign*

做電子病歷簽章動作時使用的函式。設計此函式流程(圖 4-2)**，輸入位元組陣列和** PrivateKey **物件分別為電子病歷和私密金鑰，輸出位元組陣列為電子病歷簽章。**

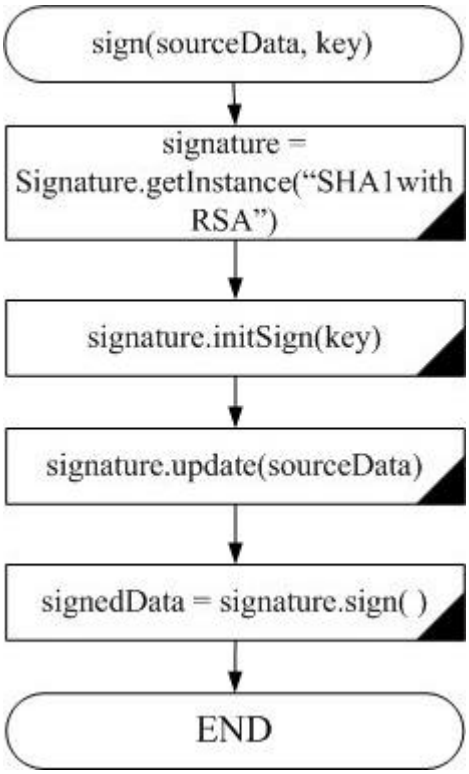

圖 4-2　sign( )**函式流程圖**

2. **電子病歷驗證程式**

這個程式分別提供 createCertificate( )、createPublicKeyFromKeyFile( )、createPublicKeyFromCertFile( )**和** verify( )**共四個函式給其他函式使用。函式說明如下：**

■ *createCertificate*

電子病歷驗證前，利用此函式先將驗證用的憑證轉換成 java.security.cert.Certificate **物件，方便要驗證簽章時取出解密用的** PublicKey **物件用。設計此函式流程(圖** 4-3**)，輸入字串為憑證在系統裡的路徑，輸出物件為** java.security.cert.Certificate**。**

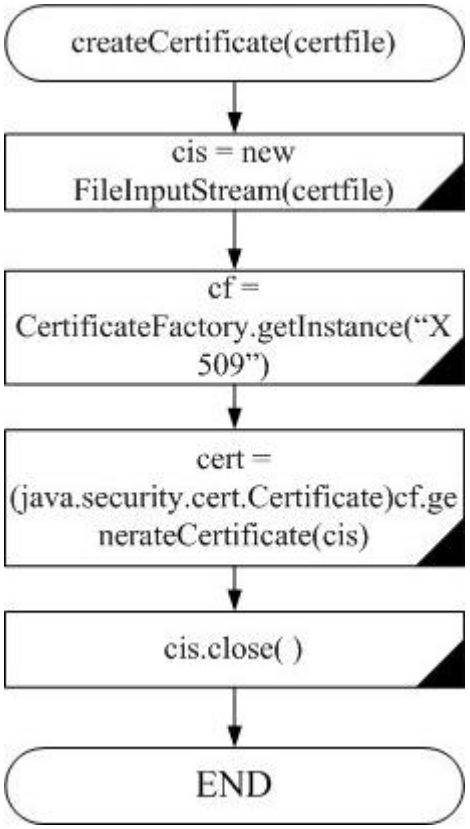

圖 4-3　createCertificate( )**函式流程圖**

■ *createPublicKeyFromKeyFile*

此函式負責將公開金鑰檔案轉換成 PublicKey **物件。設計此函式流程(圖** 4-4**)，輸入字串為公開金鑰在系統裡的路徑，輸出物件為** PublicKey**。**

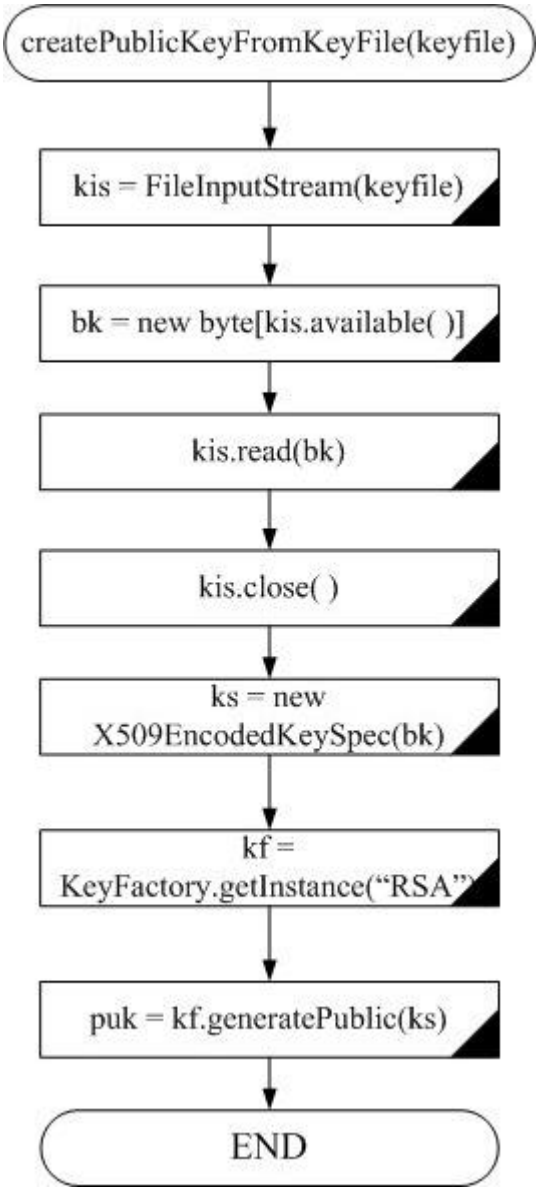

圖 4-4　createPublicKeyFromKeyFile( )**函式流程圖**

■ *createPublicKeyFromCertFile*

此函式負責從 java.security.cert.Certificate **憑證裡取出** PublicKey **物件。設計此函式流程**(**圖** 4-5)**，輸入物件為** java.security.cert.Certificate**，輸出物件為** PublicKey**。**

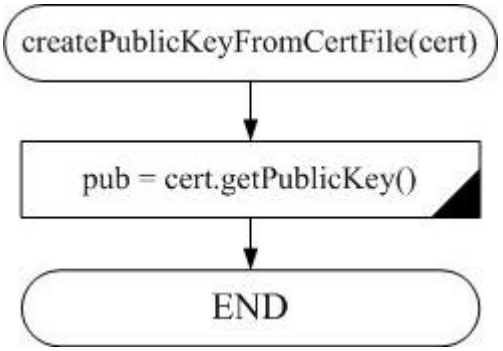

圖 4-5　createPublicKeyFromCertFile( )**函式流程圖**

■ *verify*

做電子病歷驗證動作時使用的函式。設計此函式流程(圖 4-6)**，輸入兩個位元組陣列和一個** PublicKey **物件分別為電子病歷原文、電子病歷簽章和公開金鑰，然而，此函式沒有輸出任何資料，只會在終端機視窗裡顯示驗證成功與否。**

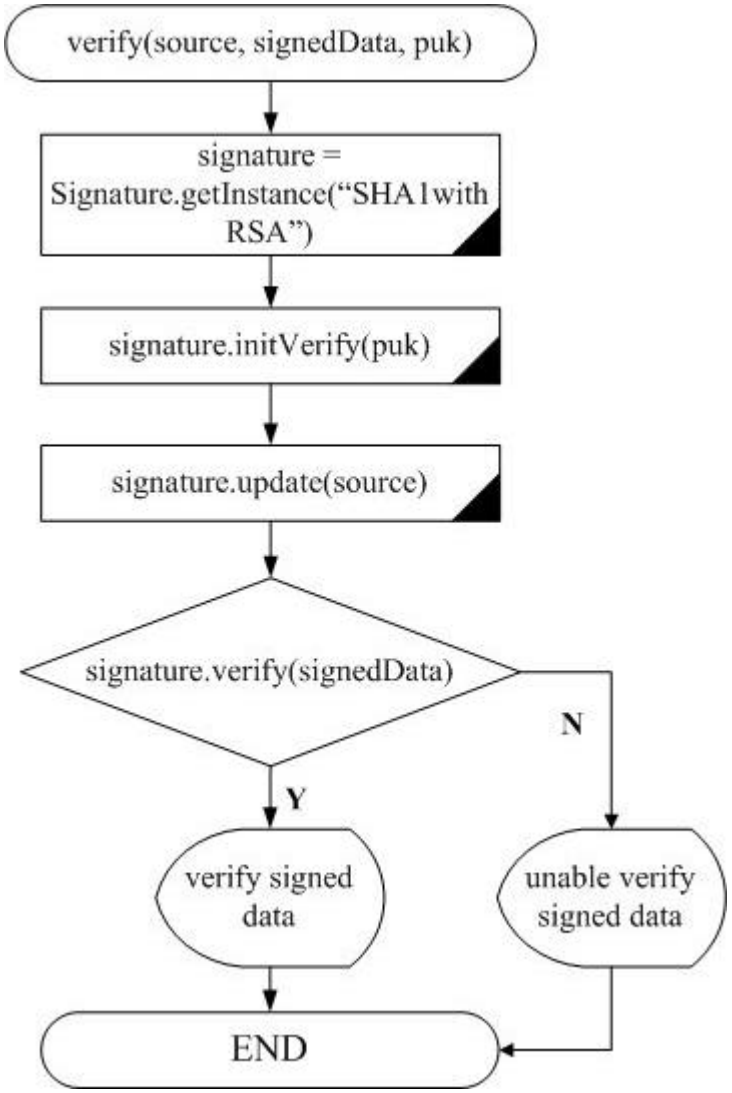

圖 4-6 verify( )函式流程圖

二、SSL 客戶端連線程式模組

1. SSL 客戶端連線程式

通常 SSL 的連線是使用者藉由瀏覽器連線到伺服器時使用的資料安全傳遞方式，而 SSL 連線時所需的交握協定、身分驗證和資料加密傳輸等，瀏覽器都會自動地幫使用者和伺服器端完成。然而，在本論文之系統中，當應用伺服器向電子病歷索引中心或

其他醫療院所的電子病歷資料庫伺服器提出 SSL 連線要求的時候，變成了伺服器和伺服器之間的 SSL 連線，此時就需要一個程式來達成這個需求，因此設計了 SSL 客戶端連線程式幫助伺服器向另一伺服器做 SSL 連線要求。

由於並不曉得未來 HCA 發行的憑證的格式及內容，因此這個程式在建立連線時並不會傳送自己的憑證給伺服器端，而是在連線建立完成後，SSL 連線要求端利用自己的私密金鑰對資料存取要求做簽章並附上憑證一起傳給接收端，接收端才驗證收到的憑證和簽章確定連線要求端是合法且可信任的：同樣的，當接收端回應訊息或資料時，也必須對回應的訊息或資料作簽章並且附上憑證，連線要求端再對收到的回應訊息或資料作驗證的動作以確保發送回應資料的來源是合法且可信任的。

2. 程式實作細節

為了將 SSL 環境初始化成不會傳送憑證給伺服器端，程式一開始先設定一個實作 X509TrustManager 介面的 class — RelatedX509TrustManager，其中程式將 X509TrustManaer 的三個函式實作如下：getAcceptIssuers( )只回傳 null 並不作任何事，checkClientTrusted( )和 checkServerTrusted( )不回傳任何值且不作任何事，接著在主程式 SSL_client_nocert( )裡，先利用 SSLContext

物件將 SSL 的環境初始化，除了指定採用 SSL 3.0 的協定外，並更改預設的 Key 管理員(KeyManager) 為 null 值以及信任管理員(TrustManager)為前面設定過的 RelatedX509TrustManager 類別，然而為了透過 SSLSocket 作輸出和輸入串流資料的方式，於是更改 HttpsURLConnection 預設的 SSLSocketFactory 為 SSLContext 在初始化 SSL 環境後取得的 SocketFactory。

在 SSL 環境初始化後，接著建立一 URL 物件，傳入想要連線的 SSL Server 的網址，然後利用 HttpsURLConnection 取得 URL 指向資源的內容型態，但是為了達成程式傳遞並接受資料的目的，程式中修改了 HttpsURLConnection 的相關參數，包括設定是否會從指定的 URL 輸入和輸出的函式 setDoInput( )和 setDoOutput( )為 True，以及設定傳送 HTTP 的 POST 網頁請求給伺服器等，於是，在建立好和伺服器連線的環境後，和一般的網路程式一樣，傳送資料時，透過 PrinterWriter 這物件及其函式傳送資料給伺服器；在接受資料時，透過 BufferedReader 這物件及其函式接受從伺服器傳回來的資料。程式流程圖如下所示：

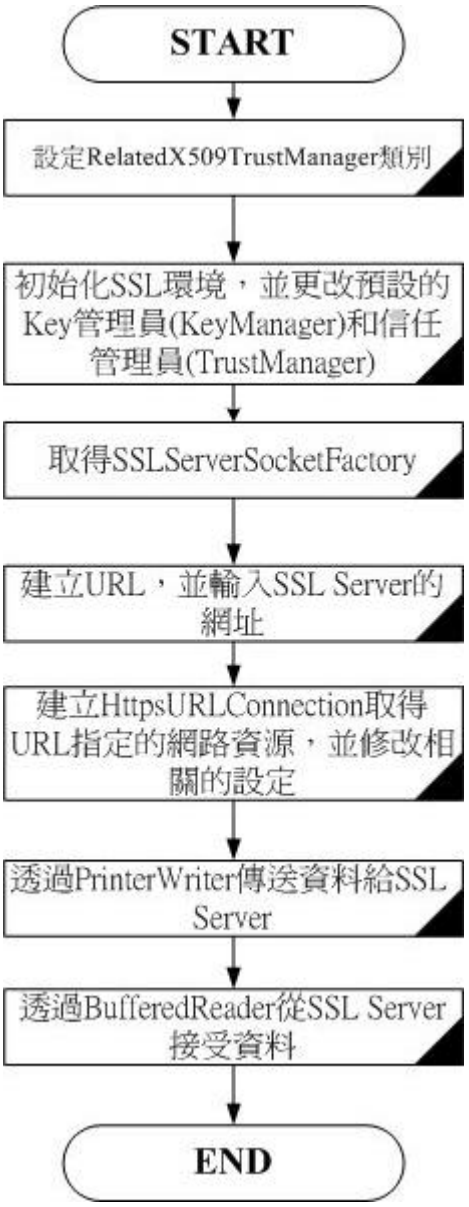

圖 4-7　SSL 客戶端連線流程圖

第伍章　結論與未來還可以加強的安全機制

第一節 結論

這次的研究成功的設計並實作出電子病歷簽章驗證程式模組和 SSL 客戶端連線程式模組，並應用在電子病歷分享系統上。 電子病歷簽章驗證程式模組整合簽章製作和驗證所需要的所有函式，透過函式的呼叫，便完成電子病歷簽章和病歷簽章驗證的動作，同樣的，透過 SSL 客戶端連程式模組的呼叫，輸入要連線的伺服器網址、要傳遞的資料，便可以和伺服器開啟 SSL 連線，安全地傳遞資料。

實作時使用的函式庫是 Java 有關安全的函式庫，包括 Java Security API 和 JSSE，這兩個函式庫提供了許多的物件與函式來方便撰寫安全方面的程式，另外，也提供了相當多的實作演算法可以利用。

往後大家利用 Java 撰寫安全模組之前，可以先參考本論文，以避免重複的錯誤嘗試。

最後，要附註的一點是，在民國九十二年四月行政院衛生署通過的「一個快速且安全的電子病歷分享模式」計劃結束之後，應用上安全技術的電子病歷分享系統已經正式於台中榮民總醫院、嘉義榮民醫院、國軍台中總醫院上線並運作正常。

第二節 未來還可以加強的安全機制

本論文之系統在對每份病歷做加密簽章時，用的是醫院的私密金鑰對整份病歷做一個簽章，但是由於整份病歷的產生是陸陸續續的經由數位醫師、檢驗師或藥劑師共同開立的，萬一病歷有誤要追究責任時，此時只有醫院的簽章並無法追究起，因此本論文可以加強對各醫師自己開立的病歷部分做簽章的機制，以達到有證據的責任追究目的。除此之外，本論文之系統還可以配合屬性憑證，利用屬性憑證裡定義每個使用者在系統裡存取資料的權限的資料，假使往後電子病歷分享系統開放給一般民眾使用，就可以利用屬性憑證來限制一般民眾只能查詢自己的病歷資料，而醫師能夠查詢每個他看診的病人的病歷資料，如此一來，便可達到「角色為基礎之存取控制(Role- based Access Control)」的機制，進而建構更安全的資料存取機制。

參考文獻

[1] S. K. Lee, C. H. Peng, C. H. Wen, S. K. Hunag and W. Z. Jiang, "Consulting With Radiologists Outside The Hospital Using Java," *RadioGraphics*, Vol.19, No.4, pp.1069-1076, 1999.

[2] C. W. Yang, D. J. Ma, C. M. Wang, C. H. Wen, C. S. Lo, and C. I. Chang, "Computer-Aided Diagnostic Detection System of Venous Beading in Retinal Images", Optical Engineering, Vol.39, No.5, pp.1-11, 2000.

[3] Persits Software, "Generate PKCS#7 Signatures and Envelopes," http://www.aspencrypt.com/task_pkcs7.html

[4] Kwangjo Kim, "Crypto API," http://vega.icu.ac.kr/~kkj/mil-capi.html

[5] Sun Microsystems, "How Do Digital Signatures Work?," http://developer.java.sun.com/developer/technicalArticles/Interviews/DigitalSigs/

[6] Sun Microsystems, "JAR File Specification," http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html

[7] Sun Microsystems, "Java™ Cryptography Architecture," http://java.sun.com/j2se/1.4.1/docs/guide/security/CryptoSpec.html#KeyFactoryEx

[8] Sun Microsystems, "Java Security API Overview," http://java.sun.com/docs/books/tutorial/security1.1/overview/index.html

[9] Sun Microsystems, “Using the Security API to Generate Public and Private Keys,” http://java.sun.com/docs/books/tutorial/security1.1/api/index.html

[10] Sun Microsystems, “Java™ Secure Socket Extension (JSSE),” http://java.sun.com/products/jsse/

[11] O’Reilly & Associates, “Secure Your Sockets with JSSE,” http://www.onjava.com/pub/a/onjava/2001/05/03/java_security.html

[12] Sun Microsystems, “Java Developer Connection - Forums,” “Topic: Getting a "peer not authenticated" Exception using Openssl,” http://forum.java.sun.com/thread.jsp?forum=2&thread=300242

[13] Sun Microsystems, “Java Developer Connection - Forums,” “Topic: SSL with client authentication using JSSE does not work.,” http://forum.java.sun.com/thread.jsp?forum=2&thread=145357

[14] Netscape Communication Corporation, “Introduction to SSL,” http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

[15] RSA Security, “SSL Basics for Internet Users,” http://www.rsasecurity.com/standards/ssl/basics.html

[16] Jupitermedia Corporation, “SSL,” http://www.webopedia.com/TERM/S/SSL.html

[17] CMP Media LLC, “OpenSSL the Cryptography Lego(TM) Set,”

http://www.unixreview.com/documents/s=2428/uni1025067917864/

[18] Sun Microsystems, “I/O: Reading and Writing (but no 'rithmetic),” http://java.sun.com/docs/books/tutorial/essential/io/index.html

[19] **行政院衛生署**, “**九十一年度醫療院所病歷電子化試辦計劃書**,” May 2002

[20] Scott Oaks, “Java™ Secuirty 2/e,” O’REILLY, Feb 2002

[21] David Flanagan, “Java Examples In A Nutshell 2/e,” O’REILLY, May 2002

[22] Aaron E. Walsh, “Foundations of Java programming for the World Wide Web,” **松崗電腦圖書資料股份有限公司**, Jan 1997

附錄一：電子病歷簽章驗證程式原始碼

1. 電子病歷簽章程式

```
package tw.gov.vghtc.exchange.cert;

import java.io.*;
import java.math.*;
import java.security.*;
import java.security.spec.*;

/**
 * 簽章服務
 */
public class SignService
{
/**
 * SignService 建構子註解。
 */
      public SignService ()
      {
      }
/**
 * 取得 PrivateKey
 * @return prk java.security.PrivateKey
 */
      public PrivateKey createPrivateKey(String keyfile)
      {
            try
            {
                  FileInputStream kis = new FileInputStream(keyfile);
                  byte[] bk = new byte[kis.available()];
                  kis.read(bk);
                  kis.close();
```

```
            KeyFactory kf = KeyFactory.getInstance("RSA");
            PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bk);
            PrivateKey prk = kf.generatePrivate(ks);
            return prk;
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
/**
 *  將資料 sourceData 以 privatekey 做簽章，只取得純簽章
 * @return byte[]
 * @param sourceData byte[]
 * @param key PrivateKey
 */
    public byte[] sign(byte[] sourceData, PrivateKey key)
        throws SignatureException, InvalidKeyException
    {
        try
        {
            Signature signature = Signature.getInstance("SHA1withRSA");
            signature.initSign(key);
            signature.update(sourceData);
            byte[] signeddata = signature.sign();

            return signeddata;
        }
        catch(Exception e)
        {
```

```
                e.printStackTrace();
                return null;
            }
        }

}
```

2. 電子病歷驗證程式

```
package tw.gov.vghtc.exchange.cert;

import java.io.*;
import java.math.*;
import java.security.*;
import java.security.cert.*;
import java.security.spec.*;
/**
 * 驗證簽章的服務
 */
public class VerifyService
{
/**
 * VerifySerivde 建構子註解。
 */
      public VerifyService ()
      {
      }

/**
 * 取得 Certificate
 * @return cert java.security.cert.Certificate
 */
      public java.security.cert.Certificate createCertificate(String certfile)
```

```
    {
        try
        {
            FileInputStream cis = new FileInputStream(certfile);
            CertificateFactory cf = CertificateFactory.getInstance("X509");
            java.security.cert.Certificate cert =
(java.security.cert.Certificate)cf.generateCertificate(cis);
            cis.close();
            return cert;
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
/**
 * 從金鑰檔案取得 PublicKey 物件
 * @return puk java.security.PublicKey
 */
    public PublicKey createPublicKeyFromKeyFile(String keyfile)
    {
        try
        {
            FileInputStream kis = new FileInputStream(keyfile);
            byte[] bk = new byte[kis.available()];
            kis.read(bk);
            kis.close();
            KeyFactory kf = KeyFactory.getInstance("RSA");
            X509EncodedKeySpec ks = new X509EncodedKeySpec(bk);
            PublicKey puk = kf.generatePublic(ks);
            return puk;
        }
```

```
            catch(Exception e)
            {
                e.printStackTrace();
                return null;
            }
        }
/**
 * 取得憑證檔案取得 PublicKey 物件
 * @return puk java.security.PublicKey
 */
        public PublicKey createPublicKeyFromCertFile(String certfile)
        {
            try
            {
                FileInputStream cis = new FileInputStream(certfile);
                CertificateFactory cf = CertificateFactory.getInstance("X509");
                java.security.cert.Certificate cert = (java.security.cert.Certificate)
cf.generateCertificate(cis);
                cis.close();
                PublicKey puk = cert.getPublicKey();
                return puk;
            }
            catch(Exception e)
            {
                e.printStackTrace();
                return null;
            }
        }
/**
 * 驗證資料的簽章, 並回傳原始資料.
 * @return byte[]
 * @param signedData byte[]
 * @param sourceData byte[]
```

```
* @param key PublicKey
 */
    public boolean verify(byte[] signedData, byte[] sourceData, PublicKey key)
        throws InvalidKeyException, SignatureException
    {
        try
        {
            Signature signature = Signature.getInstance("SHA1withRSA");
            signature.initVerify(key);
            signature.update(sourceData);
            if(signature.verify(signedData))
            {
                System.out.println("verify signed data.");
            }
            else
            {
                System.out.println("unable verify signed data.");
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return false;
        }
    }
```

附錄二：SSL 客戶端連線程式原始碼

```
package tw.gov.vghtc.exchange.source;

import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import javax.net.ssl.*;

class RelatedX509TrustManager implements X509TrustManager
{
	public java.security.cert.X509Certificate[] getAcceptedIssuers()
	{
		return null;
	}

	public void checkClientTrusted(java.security.cert.X509Certificate[] chain, String
authType) {}
	public void checkServerTrusted(java.security.cert.X509Certificate[] chain, String
authType) {}
}

public class SSL_client_nocert
{
	String urlString;

	public SSL_client_nocert(String urlString)
	{
		this.urlString = urlString;
	}

	public void run(String data) throws Exception
	{
		try
		{
			/*
			 *  SSL 環境設定初始化，並更改預設的 Key 管理員(keyManager)
```

和信任管理員(TrustManager)

```
                 */
                SSLContext ctx = SSLContext.getInstance("SSLv3");
                TrustManager[] tm = {new RelatedX509TrustManager()};
                ctx.init (null, tm, null);
                HttpsURLConnection.setDefaultSSLSocketFactory
(ctx.getSocketFactory());

                /*
                 *  建立一 URL 物件, 輸入想要連線的 SSL Server
                 */
                URL url = new URL("https", urlString, 443);
                HttpsURLConnection urlc = (HttpsURLConnection)
url.openConnection();
                urlc.setDoInput(true);
                urlc.setDoOutput(true);
                urlc.setUseCaches(false);
                urlc.setRequestProperty("server-protocol", "HTTP/1.1");
                urlc.setRequestProperty("Content-Type",
"application/x-www-form-urlencoded");
                urlc.setRequestMethod("POST");
                urlc.connect();

                /*
                 *  傳送資料給 SSL Server
                 */
                PrintWriter pw = new PrintWriter((new
DataOutputStream(urlc.getOutputStream())));
                String postParam = "index="+URLEncoder.encode(data);
                pw.print(postParam);
                pw.flush();
                pw.close();

                /*
                 * 接收 SSL Server 傳來的資料
                */
                BufferedReader from_server = new BufferedReader(new
InputStreamReader(urlc.getInputStream()));
```

```
                String line;
                while((line = from_server.readLine()) != null)
                {
                    if(line.length() == 0) break;
                    System.out.println(line);
                }

                from_server.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
}
```