第 1 章

軟體安全的演化

在研究攻擊和防禦的資安技術之前，有必要稍為瞭解軟體安全饒富趣味的悠遠事跡，透過一百年來主要安全事件的扼要描述，讀者就能明白今日 Web AP 的基礎技術發展，也可以看出安全機制的演化歷程，以及眼光獨到的駭客如何找到突破或繞過安全防制的契機。

駭客行為的起源

近二十幾年來，駭客的惡行比以往更受到關注，對駭客的來龍去脈不甚瞭解者，認為駭客行動是近 20 年才出現的，而且與網際網路密不可分。

這些看法不全然正確。不可否認，駭客攻擊頻率的確隨著全球資訊網的興起而呈指數增長，但駭客活動早在 20 世紀中葉就已經出現了，若以不同角度定義「駭客活動」，則出現的時間甚至更早。許多專家不斷爭論現代駭客真正出現的年代，然而，1900 年初期的一些重大事件與現今看到的駭客活動亦極為相似。

1910 及 1920 年代的一些特殊事件也可能被視為駭客活動，其中多數涉及竄改摩斯電碼的收發內容或干擾無線電傳輸，雖然確有其事件，但這類攻擊並不常見，且攻擊行為很難造成大規模影響。

筆者身為資安專家，主要工作是為企業的底層架構及程式碼問題尋找解決方案，在成為資安專家之前，曾從事多年軟體開發工作，能夠以多種程式語言和框架撰寫 Web AP，時至今日仍以安全性自動化的形式撰寫軟體，也在業餘時間參與各種專案。筆者並非歷史學家，沒有打算在本節爭論駭客活動的細節或起源，相反地，希望由筆者多年的獨立研究結果，從這些事件中汲取經驗並套用於今日的形勢上。

本章並不打算全面檢視駭客活動，只會參考一些重要的歷史事件。讓我們研究一些有助於塑造當今駭客與軟體工程師關聯性的歷史事件，不要再去糾結無關緊要的旁枝末節。現在，且將時間點拉回 1930 年代初期。

1930 年代，謎機

恩尼格瑪密碼機（Enigma machine，如圖 1-1）又稱為**謎機**，使用電氣機械轉輪加密和解密無線電傳輸的明文資料，此設備是德國人發明的，對於第二次世界大戰而言，這可是相當重要的技術進展。

圖 1-1：恩尼格瑪密碼機

謎機外觀就像大型矩箱或矩形機械式打字機，每次按下鍵盤時，這些轉輪就會配合轉動並得到一個看似隨機的字元，這些字元隨後會傳送給下一個或多個接收點的謎機。然而，這些字元的產生並非真正隨機，而是受轉輪旋轉及設備上可隨時修改的設定選項所決定，任一台具有對應設定的謎機都能讀取或「解密」另一台謎機發送過來的訊息，為避免交換的機密訊息被攔截，謎機提供極大幫助。

第 3 章

Web 應用系統的結構

在評估如何進行有效的 Web 應用系統偵查之前，最好先瞭解 Web 應用系統常用的第三方元件之共通技術，第三方元件包括 JavaScript 程式庫和預先定義的 CSS 模組，以及應用系統所在的 Web 伺服器，甚至伺服器的作業系統。清楚各個元件扮演的角色，以及它們在架構堆疊上所發揮的功效，便可快速輕鬆地識別它們及找出設定上的缺失。

Web 應用系統的前世今生

今日 Web 應用系統所使用技術，已非 10 年前可比擬，建置 Web AP 的工具比以前進步太多了，甚至會讓人覺得已整個脫胎換骨，令人耳目一新！

十多年前，多數 Web 應用系統以伺服器端框架建構，由伺服器產生 HTML ＋ JS ＋ CSS 所組成的頁面，再傳送到使用者端；當需要更新呈現內容時，使用者端只需透過 HTTP 管線向伺服器請求另一組頁面，再依照接收到的內容進行渲染（繪製畫面）。

但沒過多久，隨著 Ajax 的興起，Web 應用系統便頻繁地使用 HTTP 協定，讓連線階段（Session）的頁面裡之 JavaScript 可以向伺服器發出請求。

如今，許多應用系統是透過網路與兩個以上其他系統通信，以便建構出更優質的畫面，不再由單一系統供應所有素材，這是現今 Web 應用系統與十年前系統在架構上的主要差異。

現今的 Web 應用系統常常利用表現層狀態轉換（REST）API 連接數個應用程式組合而成的，API 本身並無狀態，只用於滿足應用程式對另一個應用程式的請求，亦即，它們並不會儲存請求者的任何訊息。

用戶端（UI）程式則以類似傳統視窗程式的形式在瀏覽器中運行，用戶端程式管理自己的生命週期循環、請求自己所需的資料，在完成頁面初始化之後，就不需要一再重新載入後續頁面。

配送到 Web 瀏覽器的獨立應用程式可與多部伺服器通訊，其實沒什麼好大驚小怪的，想像一支需要使用者登入的圖片管理程式，它在某個 URL 上可能有一部專門管理及發送圖片的伺服器，另一個 URL 上的伺服器則負責資料庫管理和使用者登入。

可以肯定地，現今的應用系統是由許多獨立，但彼此共生的系統協同合作而成，會如此發展，主要是因為有明確定義的網路協定和 API 架構。

常見的 Web 應用系統可能使用以下幾種技術：

- REST API
- JSON 或 XML
- JavaScript
- SPA 框架 (如 React、Vue、EmberJS 或 AngularJS)
- 身分驗證和授權系統
- 一台以上的 Web 伺服器（通常架在 Linux 伺服器上）
- 一套以上的 Web 服務平台（ExpressJS、Apache 或 NginX）
- 一套以上的資料庫系統（MySQL、MongoDB 或其他)
- 用戶端的本機資料儲存體（cookies、web storage 或 IndexDB）

這裡所列的並非全部技術，畢竟網際網路上有數十億個獨立網站，根本不可能用一本書涵蓋所有 Web 應用系統的技術。

讀者若需要快速掌握未列於本章的特定技術，可閱覽其他書籍和有關程式開發的網站，例如 Stack Overflow。

某些技術在十年前就有了，不過，這段時間裡又有不小改進。像資料庫就已存在幾十年，但 NoSQL 資料庫和用戶端資料儲存體則是之後才發展出來的；又如，直到 NodeJS 和 npm 出現後，全部以 JavaScript（以下簡寫成 JS）開發 Web 應用系統的技術才被迅速採用。近十年來，Web 應用系統格局變化之快，某些原本沒沒無名的技術因而暴紅。

第 5 章

API 分析技巧

在發現子網域之後，下一個偵查技能便是 API 端點分析，應用系統會用到哪些網域？如果牽涉此應用系統的網域有三個（如 x.domain、y.domain 和 z.domain），應注意它們可能都有自己的獨特 API 端點。

這裡使用的技術與查找子網域所用的技術相似，暴力探索和字典檔探索也有很好效果，就算手動探索和利用邏輯分析，也能得到不錯回報。

對於研究 Web 應用系統結構，查找 API 是繼子網域探索之後的下一步，此步驟可提供理解 API 功用所需資訊，知道 API 為何要向網際網路公開之後，便能瞭解它與應用系統的關係及其功能目標。

探索 API 端點

前面說過，當今多數企業應用系統在定義 API 結構時會遵循特定方案，通常是遵循 REST 或 SOAP 格式，現在，REST 變得更受歡迎，也被認為是 Web 應用系統 API 之理想架構。

使用瀏覽器的開發人員工具探索應用系統和分析網路請求時，若發現許多類似下列的 HTTP 請求：

```
GET api.mega-bank.com/users/1234
GET api.mega-bank.com/users/1234/payments
POST api.mega-bank.com/users/1234/payments
```

身分驗證機制

要猜測 API 端點所需的載荷形式，遠比評斷既存的 API 端點支援哪些動詞要困難得多。

最簡單的方法就是透過瀏覽器發送已知功用的請求，然後分析此請求的結構，除此之外，還要根據其他情報去猜測 API 端點的載荷形式，並手動測試這些載荷形式是否有效。雖然可以利用自動化方式探索 API 端點的結構，但不做任何分析就去嘗試發送 API 請求，很容易被對方偵測和記錄到日誌裡。

最好下手的地方應該是每個應用系統都能找到的通用端點：登入、註冊及重設密碼等，這些端點在每個應用系統使用的載荷形式都很相像，因為身分驗證一般會依照標準邏輯來設計。

每個具有公開的 Web 使用者界面之應用系統，應該都會有一組供使用者登入（login）的頁面，儘管彼此的登入頁面稍有不同，但其功能都是用來驗證連線身分。因為應用程式會在每次請求時一併發送身分驗證**符記**（token），瞭解應用系統使用的身分驗證方案就顯得重要，如果可以利用逆向工程找出身分驗證的類型，並瞭解符記是如何附加在請求封包，則分析依賴此身分驗證符記之其他 API 端點，將會變得更加容易。

目前有幾種主要的身分驗證方案，最常用的認證方案如表 5-2 所示。

表 5-2：主流的身分驗證方案

身分驗證方案	實作細節	優點	缺點
HTTP 基本驗證	每次請求會發送 Base64 編碼後的帳號：密碼資料	所有主流瀏覽器都支援	Session 不會過期，容易遭到竊聽
HTTP 摘要驗證	每次請求會發送雜湊後的帳號：領域：密碼資料	較不易被竊聽；伺服器可以拒絕過期的身分符記（token），	加密強度受使用的雜湊演算法限制
OAuth	以「Bearer」符記為基礎的驗證機制；可以由另一個網站執行登入驗證，例如使用者由 Amazon 登入，Twitch 則透過 Amazon 驗證使用者身分	符記化的權限機制，可以在不同應用系統間分享身分驗證結果，進而達到整合目的	可能存在網路釣魚風險；只要負責管理身分的中心站台被入侵，其他相關應用系統也會有被入侵的可能

第 7 章

尋找應用系統架構的弱點

截至目前已經介紹許多技術，可應用於識別 Web 應用系統使用的組件、確認 API 端點的形式，以及熟悉 Web AP 如何與瀏覽器互動，每一種技術都有它的價值，若能將它們收集來的資訊有條理地組合在一起，會創造出更高的價值。

最好能像之前提過的，在整個偵查過程中，將得到的情報以某種形式記錄下來。某些 Web 應用系統非常龐大，可能需要花幾個月時間才能完整調查，因此，將研究結果作成文件是不可或缺的步驟，偵查過程中到底需要記錄多少資料？可依個人（測試人員、駭客、業餘愛好者或工程師等）需求而定，雖說資料寧多勿缺，但未事先區分優先等級，有時記錄大量資料未便能產生等量價值。

對於所測試的每套應用系統，最好提供井然有序的說明，內容至少包括：

- Web 應用系統使用的技術。
- 依照 HTTP 動詞列出可用的 API 端點。
- 列出 API 端點形式（如果有）。
- Web 應用程系統所包含的功能（如留言、身分驗證、訊息通知等）。
- Web 應用系統使用的網域。
- 所有找到的組態資訊，例如內容安全性原則（CSP）。
- 身分驗證／ Session 管理機制。

一旦整理完成這分清單後，便可利用它安排查找弱點或攻擊漏洞的先後順序。

筆者與其他作者有不同看法，Web 應用系統裡的多數漏洞源自設計不良的系統架構，而非不當的程式撰寫習慣。直接將使用者提供的 HTML 寫入 DOM，這種處理方法絕對有風險，若未適當清理，使用者便可藉此上傳腳本，讓腳本在另一位使用者的電腦上執行（XSS 攻擊）。

同樣的架構，在其他地方的應用系統照樣出現一大堆 XSS 漏洞，但同一行業中其他具有相當規模的應用系統卻幾乎沒有 XSS 漏洞，為什麼呢？追根究柢，應用系統的架構及該系統引入的第三方模組或元件的架構竟是弱點的源頭，而漏洞就是由這些弱點造成的。

從架構信號研判安全與否

誠如前述，單一 XSS 事件可能是程式撰寫不當造成的，當出現許多漏洞時，就很可能是應用系統架構脆弱的信號。

想像有兩支簡單的應用程式，它們都允許使用者傳送一般文字訊息給另一位使用者（即時通訊），但其中一支有 XSS 漏洞，另一支卻沒有。

當使用者透過 API 端點請求儲存文字訊息時，不安全的這一支程式可能沒有拒絕腳本文字，文字訊息並沒有得到適當的過濾和清理，腳本文字被存入資料庫，最後，訊息被加入 DOM 裡，經 DOM 評估成「*test message<script>alert('hacked');</script>*」，從而導致腳本被執行。

另一邊，安全的應用程式可能具有層層防護。要在每個案件都實作多層次保護，對開發工程而言相當耗費時間，也容易被疏忽。

如果應用系統的架構原本就不安全，即使由具有程式安全技能的工程師開發應用程式，最終仍然可能出現安全漏洞，此乃因安全性高的應用程式，是從實作之前及開發當下就融入安全條件；安全性中等的應用程式，是在開發過程中加入安全防護；安全低的應用程式則可能不實作任何安全功能。

如前述即時通訊（IM）系統的開發人員須在 5 年內撰寫出 10 個版本，每個版本的實作方式可能會有所不同，但各個版本面臨的安全風險卻是相近的。

IM 系統都會包含下列功能：

- 提供編寫訊息的 UI。
- 用來接收使用者剛剛書寫及提交的訊息之 API 端點。

第 10 章

跨站腳本（XSS）

跨站腳本（XSS）是網際網路上常見漏洞之一，隨著使用者與 Web 應用系統互動頻率升高，XSS 威脅也不斷增加。

就本質而言，XSS 攻擊係因使用者的瀏覽器執行 Web 應用系統上之腳本所引起，若能以任何方式（尤其由終端用戶）污染或竄改腳本，讓動態建立的腳本交由瀏覽器執行，Web 應用系統便會面臨危機。

XXS 攻擊主要分成三大方式：

- 儲存型（stored XSS；程式碼被執行前，會先儲存於資料庫裡）。
- 反射型（reflected XSS；程式碼沒有儲存在資料庫裡，而是直接由伺服器反彈回來）。
- DOM 型（DOM-Based XSS；程式碼由瀏覽器保存並執行）。

雖然還有一些 XSS 類型是在這三種分類之外，但這三大分類已涵蓋現今 Web 應用系統的絕大多數 XSS，一些資安社群，如**開放網路應用安全計畫**（OWASP）等都將 XSS 攻擊向量視為 Web 上的常見威脅。

進一步討論這三種類型 XSS 之前，先來看看 XSS 的成因，以及讓此類攻擊生效的系統缺失。

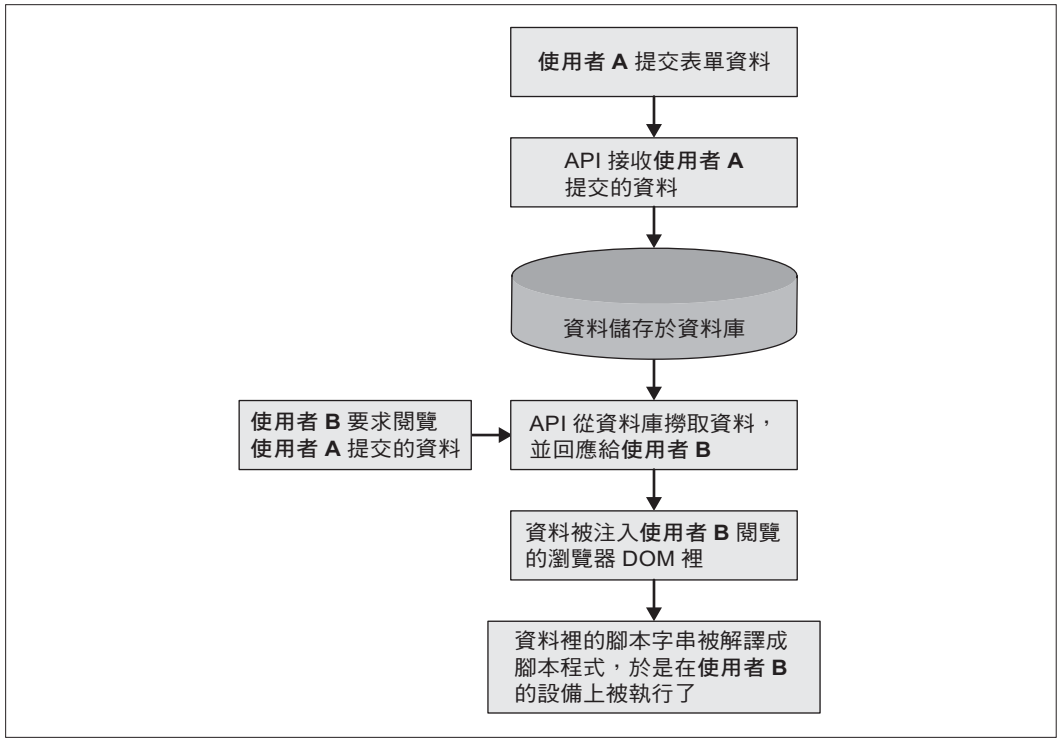

圖 10-1：儲存型 XSS 是使用者上傳的惡意腳本被儲存在資料庫裡，其他使用者請求並查看時，就在其電腦上執行此腳本

儲存在資料庫的物件可以被許多使用者瀏覽，某些情況下，全域物件遭到感染後，會讓所有使用者都暴露在儲存型 XSS 的攻擊範圍。

假設讀者負責維運或維護某一影音管理網站，首頁會呈現使用者「推薦」的影集，該影集標題存在儲存型 XSS 攻擊，則在影集下架之前，可能危害此網頁的每位訪客。正因如此，儲存型 XSS 攻擊才會對機構形成致命威脅。

另一方面，儲存型 XSS 持續存在的特性，讓它很容易被檢測到，雖然腳本在用戶端（瀏覽器）執行，卻儲存在資料庫（或稱伺服器端）上。腳本以純文字方式儲存在伺服器端，但不會被伺服器執行。除非是使用 Node.js 伺服器的特殊情境，在這種情況，被執行的腳本程式碼將歸為**遠端程式碼執行**（RCE），後面會介紹這種攻擊。

第 11 章

跨站請求偽造（CSRF）

有時雖已知道有一個 API 端點可以執行我們所期望的操作，但囿於權限不足（例如需管理員帳戶）而無權調用該端點。

本章將討論如何進行**跨站請求偽造**（CSRF）攻擊，在不使用 JavaScript 腳本的情況下，讓管理員或特權帳戶代替我們執行想要的操作。

CSRF 攻擊是利用瀏覽器的行為及瀏覽器與網站之間的信任關係。只要能找到確保交易安全的信任關係，而瀏覽器又開放過多信任時，便可以編製一些鏈結和／或表單，並花點力氣，想辦法讓使用者在不知情的情況下，透過這些鏈結或表單發送請求。

由於是瀏覽器在背景發送請求，被攻擊的使用者通常不會查覺到 CSRF 攻擊，因此，可在不被使用者查覺的情況下，利用特權使用者的權限對伺服器發出請求，這是最隱秘的 Web 攻擊之一，自 2000 年初發跡以來就對網路造成極大浩劫。

竄改查詢參數

來看看最基本的 CSRF 攻擊形式：竄改超鏈結的參數。

網路上，多數的超鏈結表單是與 HTTP GET 請求相對應，常見的形式是嵌在 HTML 原始碼裡的「 提示文字 」。

無論從何處發送、向何處讀取或如何透過網路傳輸，HTTP GET 請求的結構依然簡單且一致，要讓 HTTP GET 有效，就必須遵守 HTTP 所規定的版本，因此不必擔心 GET 請求的結構在應用系統間會有所差異。

HTTP GET 請求的結構如下：

```
GET /resource-url?key=value HTTP/1.1
Host: www.mega-bank.com
```

每個 HTTP GET 請求都包括請求方法（GET）、緊隨其後的資源網址，然後是一組選用的查詢參數（表單資料），查詢參數的開頭會以問號（?）表示，查詢參數集最後以空白字元結尾，再來就是 HTTP 的規格。而下一列是資源網址所在的主機位址。

當 Web 伺服器收到此請求，會將它轉送給適當的處理程式，該處理程式接收查詢參數及其他資訊，判斷發出請求的使用者身分、發出請求的瀏覽器類型及此請求期待收到的資料格式。

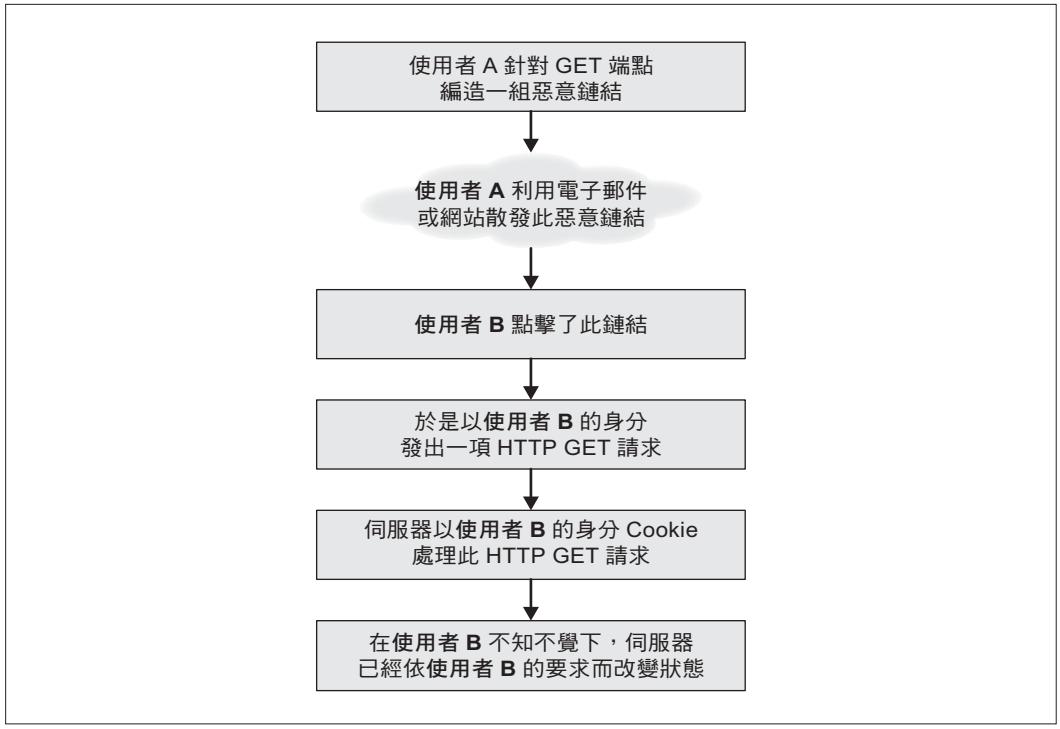

圖 11-1：散播 CSRF GET 的惡意鏈結，當使用者點擊後，便以通過身分驗證的使用者身分改變 HTTP GET 請求的狀態