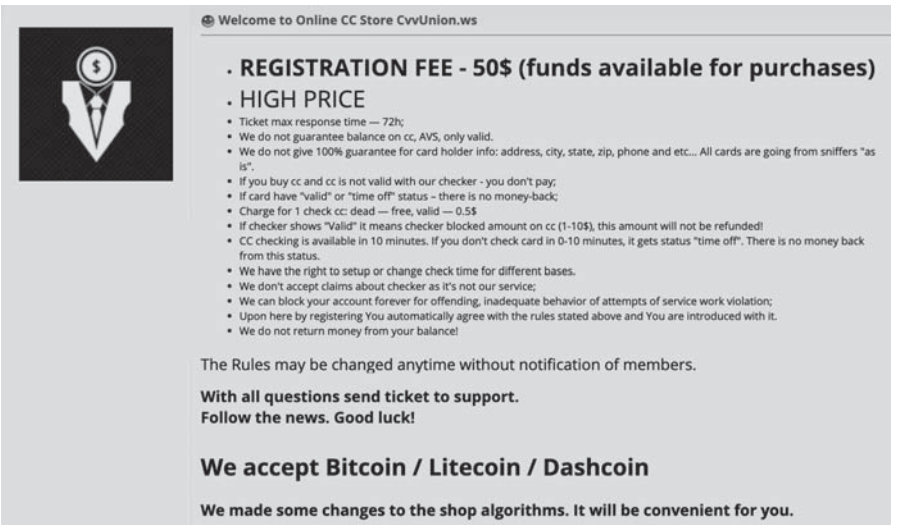


**圖 1-1：嗨！你顯然是俄羅斯的高級駭客，不是巡邏暗網的 FBI 探員，我想跟你購買一些信用卡資料**

有些免費的駭客工具可以輕易攻擊最新漏洞，甚至不必到暗網去找，透過 Google 搜索，很快就能找到想要的東西了，來看看是如何辦到的。

# 如何入侵網站

要發動入侵其實不難，作業步驟如下：

1. 到 Google 搜尋並**下載 kali linux**。Kali 是專為駭客搭建的免費 Linux 作業系統，內建 600 多套安全和駭客工具，Kali 由 Offensive Security 的安全研究人員維護。

2. 在電腦上安裝虛擬機容器。讀者可在電腦的虛擬機容器裡安裝其他作業系統，不會覆蓋目前使用的作業系統之主機環境。免費的 Oracle VirtualBox 可以安裝在 Windows、macOS 或 Linux 上，透過虛擬機容器能夠輕易安裝及執行 Kali Linux，不必經由繁瑣的設定手續。

3. 在虛擬機容器中安裝 Kali Linux。只要雙擊下載回來的 Kali Linux 安裝程式就可以啟動安裝程序。

4. 啟動 Kali Linux，然後執行 Metasploit 框架。如圖 1-2 所示，*Metasploit* 是極受資安人員及駭客喜愛的命令列工具，可用於測試網站安全性和檢查漏洞。


圖 1-2：只要擁有高超技術的 ASCII 字元乳牛，就能執行駭客攻擊。

5. 對目標網站執行 Metasploit 裡的 `wmap` 工具，查看找到哪些漏洞，執行結果類似圖 1-3。`wmap` 掃描網站的 URL，測試 Web 伺服器是否存在安全漏洞。基於法律責任，你只能在自己的網站執行這項掃描！

因為套用樣式資訊時必須遵循嚴謹的規則優先順序，渲染管線實作很多邏輯來解碼最終的樣式，每個選擇子可套用到頁面上的多個元素，每個元素通常也會由多個選擇子指定樣式資訊。網際網路日益增長所帶來的痛點之一，是要如何讓網站內容能在不同瀏覽器上呈現一致的外觀，縱使新一代的瀏覽器對網頁渲染有一致作法，但最終結果仍有所差異，Acid3 測試是判斷網頁內容是否符合 Web 標準的業界基準，如圖 3-1 所示，能夠拿到滿分的瀏覽器並不多，讀者可以到 *http://acid3.acidtests.org/* 進行 Acid3 測試，看看你的瀏覽器能拿幾分！

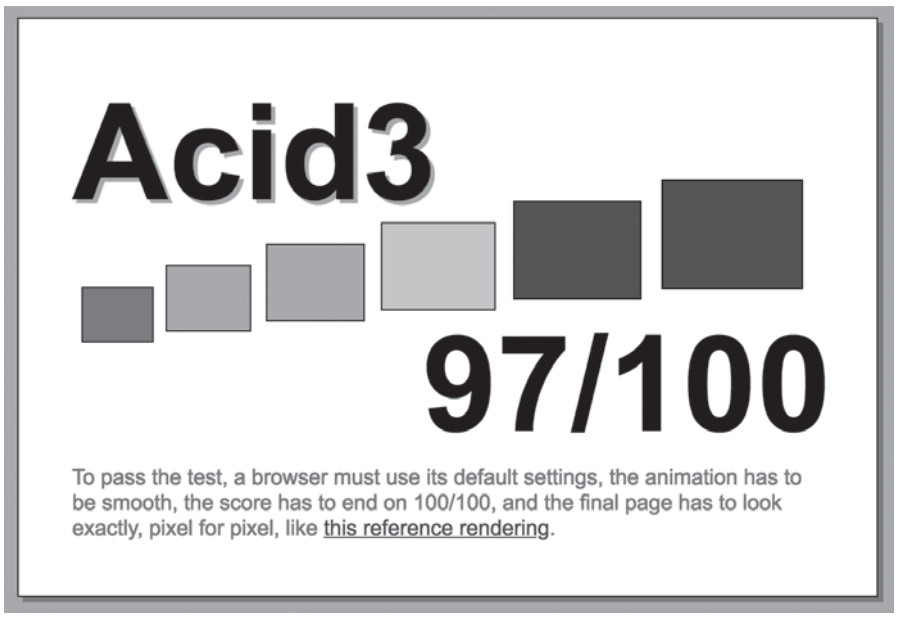


圖 3-1：Acid3 用來確認瀏覽器可否正確渲染彩色矩形

瀏覽器在建構 DOM 樹及套用樣式規則時，也會一併處理網頁裡的 JavaScript，JavaScript 甚至可以在渲染網頁面之前變更 DOM 的結構和布局，來看一下 JavaScript 怎樣搭配渲染管線執行。

經過前面的學習，相信讀者對網際網路的工作原理已有深刻理解，那就來看看特定漏洞的成因，以及駭客如何利用它們進行攻擊。本章要介紹注入攻擊，當駭客將外部程式碼注入應用程式，用以改變應用程式邏輯或讀取機敏資料時，就會造成注入攻擊。

回想一下，網際網路其實也是主從式（*client-server*）架構用例，亦即，Web 伺服器可以同時處理多用戶端的連線，多數用戶端是 Web 瀏覽器，當使用者瀏覽網站時，瀏覽器代理使用者向 Web 伺服器發出 *HTTP* 請求，Web 伺服器則回傳 *HTTP* 回應，其中包含構成網站內容的 HTML。

由於 Web 伺服器掌控網站的內容，伺服器端的程式碼自然期待與使用者產生特定形態的互動，因此，希望瀏覽器發出可預期的 HTTP 請求，例如，使用者點擊鏈結（link）時，伺服器會期待看到對 URL 的 `GET` 請求，或者使用者輸入身分憑據並點擊「確定」鈕時，會看到 `POST` 請求。

然而，瀏覽器也可能發送伺服器預想不到的 HTTP 請求形態，此外，Web 伺服器也樂意接受來自其他類型的用戶端之 HTTP 請求，並非僅限瀏覽器。

會使用 HTTP 用戶端函式庫的開發人員就可以編寫腳本，向網際網路上的任何 URL 發送請求。第 1 章提到的駭客工具就可以做這樣的事。

伺服器端的程式碼並沒有可靠的方法來判斷是誰發出 HTTP 請求，因為 HTTP 請求的內容與用戶端無關，無法依照內容來判斷瀏覽代理的身分，目前較佳的方式是檢查 `User-Agent` 標頭所攜帶的瀏覽代理之描述，但是腳本和駭客工具常會偽造此標頭內容，讓瀏覽代理看起來像是一般瀏覽器。[譯註 1]

譯註 1　舉凡以 HTTP 協定和 Web 伺服器通訊的工具，統稱為「瀏覽代理」，最常見的是 Web 瀏覽器。


圖 6-1：使用 URL 注入惡意命令

駭客利用 domain 參數提交「google.com && echo "HAXXED"」，從瀏覽器網址列可看到參數中的空格和特殊符號經過 URL 編碼處理。在 UNIX 系統可以用「&&」連接兩個獨立的命令，因為上面的 PHP 程式碼沒有刪除此類控制字元，駭客便可精心編製 HTTP 請求，以便附加一組額外的命令，在這種情況下，作業系統將執行兩個單獨的命令：預期用來查找 *google.com* 的 nslookup 命令，以及跟在 nslookup 後面所注入的「echo "HAXXED"」命令。

這個例子是注入無害的「echo」命令，只會在 HTTP 回應輸出「HAXXED」，但駭客可以使用此漏洞注入並執行伺服器上的任何命令，不需吹灰之力便能瀏覽檔案系統、讀取機敏資料，甚至破壞整個應用系統，藉由 Web 伺服器調用作業系統命令的功能，駭客幾乎可完全控制系統，除非讀者已採取嚴謹措施減輕影響程度。

利用第 6 章介紹的 SQL 注入手段就能將惡意 JavaScript 植入資料庫，不過，駭客更常透過合法途徑插入惡意程式碼，例如，網站允許使用者發表評論，而且會將該則評論儲存於資料庫，當其他使用者閱覽同一主題的評論時，瀏覽器就會顯示評論內容，在這種情況下，駭客執行 XSS 攻擊的簡單方法是提供含有「`<script>`」標籤的評論，讓 JavaScript 隨同評論的一般文字寫入資料庫。網站若未能安全建構 HTML，每當使用者瀏覽該網頁時，`<script>` 標籤都會輸出到瀏覽器，裡頭的 JavaScript 將在受害者的瀏覽器上執行。

舉個例子，假設有個受歡迎的烘焙社群網站，網址為 *https://breddit.com*，網站鼓勵用戶參與討論麵包烘焙話題，線上論壇的大部分內容都是由用戶貢獻的，每當用戶發表言論，網站就會將其內容寫入資料庫，及呈現給參與同一主題的其他使用者，於是駭客有機會利用評論注入 JavaScript，如圖 7-1 所示。

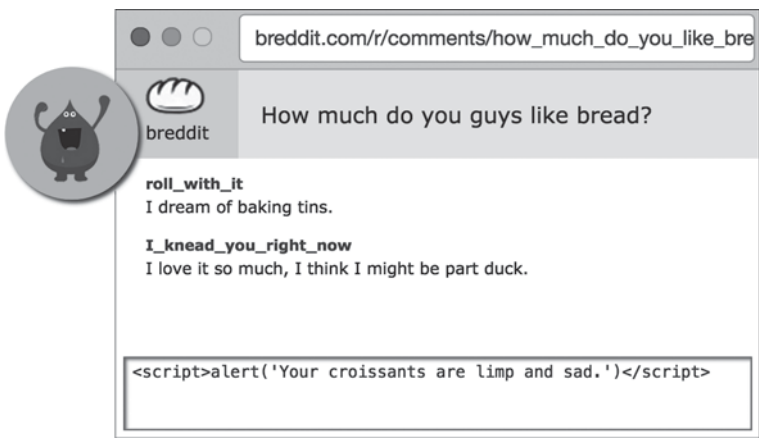


圖 7-1：駭客利用撰寫評論，在網站注入 JavaScript

假如網站在編寫 HTML 時沒有對注入的腳本做轉義處理（下一節討論），則下一位閱讀評論的用戶就會把駭客注入的 <script> 引薦給瀏覽器執行，結果如圖 7-2 所示。

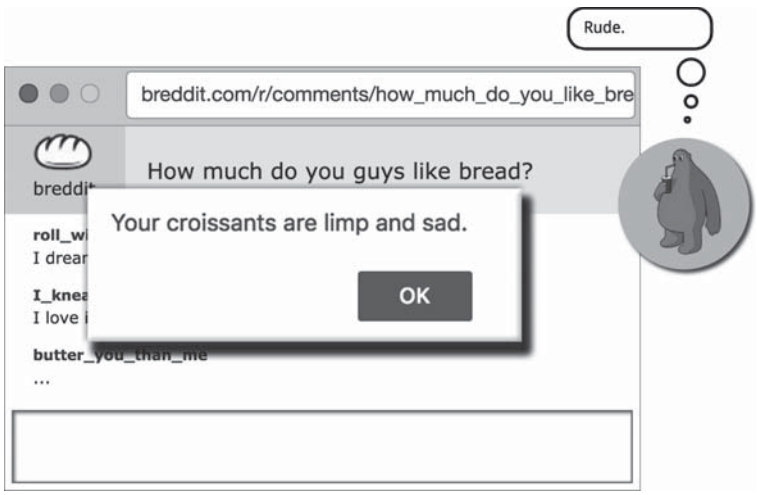


圖 7-2：駭客的 <script> 標籤被輸出給受害者的瀏覽器執行

無聊的 alert() 對話框只會干擾使用者體驗，但駭客通常用這種方法檢查是否存在 XSS 弱點，若能呼叫 alert() 函式，就可能升級成更具威力的攻擊，例如竊取其他使用者的 session，或將受害者導向惡意網站。此烘焙社群不再是安全的線上服務了！

論壇的評論並非唯一存在此類漏洞的地方，只要是用戶可控制的任何內容，都是需要被保護的潛在攻擊途徑。駭客可以在用戶名稱、個人資料頁面、線上評論等處注入惡意腳本而達到 XSS 攻擊的目的。接著來看看幾個簡單的保護手法。

## 防範措施 2 ：實作防 CSRF Cookie

解除 GET 請求變更伺服器狀態的能力，可以抵禦多數 CSRF 攻擊，但也需要防範利用其他 HTTP 的請求方法，雖然，比起 GET 請求，駭客很少使用其他請求方法發起 CSRF 攻擊，因為手續過於繁瑣，但如果能有不錯收益，駭客還是會願意嘗試。

例如，引誘受害者從受駭客控制的網站提交惡意資料表單或腳本，向我們的網頁發出 POST 請求。若網站會回應此類 POST 請求並執行敏感操作，就需要藉由防 CSRF Cookie 防範來自網站外部的請求，只允許從自家網站的登入表單和 JavaScript 觸發敏感操作，而不回應駭客引誘使用者所提交的惡意頁面。

防 *CSRF Cookie* 是 Web 伺服器寫在 Cookie 參數的隨機字串。回想一下，Cookie 是 HTTP 標頭中在瀏覽器和 Web 伺服器之間來回傳遞的一小段文字，如果 Web 伺服器回傳的 HTTP 回應包含「`Set-Cookie: _xsrf=5978e29d4ef434a1`」標頭，則瀏覽器在下一回的 HTTP 請求，會以「`Cookie: _xsrf=5978e29d4ef434a1`」形式送回相同的資訊。

安全的網站會使用防 CSRF Cookie 驗證 POST 請求是否來自同一 Web 網域上的網頁，在此網站上的 HTML 表單裡，以「`<input type="hidden" name="_xsrf" value="5978e29d4ef434a1">`」元素記錄相同的隨機字串，以便經由 POST 請求傳送給 Web 伺服器，若使用者提交給伺服器的表單，其 Cookie 的 `_xsrf` 值與表單 `_xsrf` 欄位的值不一致時，伺服器應斷然拒絕該請求，如此一來，伺服器就能驗證及確保請求是來自網站內部，而非惡意的第三方網站，因為只有相同網域載入網頁時，瀏覽器才會發送所需的 Cookie。

現在的 Web 伺服器都支援防 CSRF Cookie，由於不同 Web 伺服器之間的語法略有差異，請查閱 Web 伺服器的安全文件，確保瞭解如何實作這些 Cookie。清單 8-2 是 Tornado Web 伺服器的模板檔案，裡頭包含防 CSRF 的保護符記（token）。


圖 9-1：Google Chrome 提供的原生登入界面會中斷瀏覽器操作

## 非 HTTP 原生的身分驗證

由於 HTTP 原生身分驗證的不友善設計，大概只有與使用者體驗無關的應用程式會使用，近代網站大多另以 HTML 格式（清單 9-1）實作自己的登入表單。

```
<form action="/login" method="post">
❶ <input type="email" name="username" placeholder="Type your email">
❷ <input type="password" name="password" placeholder="Type your password">
  <input type="submit" name="login" value="Log in">
</form>
```

清單 9-1：典型的 HTML 登入表單

典型的登入表單包含使用者輸入帳號的「`<input type="text">`」元素 ❶ 和輸入密碼時會顯示「•」以遮隱密碼字元的「`<input type="password">`」元素，使用者輸入的帳號和密碼，在提交（submit）表單時以 `POST` 請求送給伺服器，若因無法驗證使用者身分而導致登入失敗，伺服器則以 HTTP 回應 `401` 狀態碼；若登入成功，伺服器會將使用者重導至主要作業頁面。

## 防範措施 1 ：使用第三方身分驗證機制

自己開發的身分驗證系統常常考慮不周延，最安全的作法是使用第三方服務，與其自己開發，不如考慮使用 Facebook 登入之類的第三方服務，讓使用者以他的社交平台之身分憑據在你的網站進行身分驗證，既可提供使用者便利服務，又能減輕網站擁有者保存帳密的負擔。

其他大型科技公司也提供類似的身分驗證服務，多數服務是以開放式身分驗證（*OAuth*）或 *OpenID* 標準為基礎，它們是將身分驗證服務委託第三方處理的常見協定，這些身分驗證系統也可以混搭使用，可以選擇一項或多項符合基礎用戶目標的服務，功能整合程序並不會太複雜。若網站也要提供電子郵件服務，可以選擇和 Google OAuth 整合，要求使用者使用 Gmail 帳戶；若要提供技術服務，可考慮使用 GitHub OAuth，也可以選用 Twitter、Microsoft、LinkedIn、Reddit、Tumblr 及其他數百家廠商提供的身分驗證服務。

## 防範措施 2 ：與單一登入整合

若是與 OAuth 或 OpenID 的身分供應商整合，你的用戶通常以個人電子郵件位址作為帳號，如果網站的目標用戶是企業內部使用者，請考慮與 Okta、OneLogin 或 Centrify 之類的單一登入（*SSO*）系統整合，讓整個企業的系統能集中處理身分驗證，讓員工以企業的電子郵件位址無縫地登入第三方應用程式，公司管理員保有員工可以存取哪些網站的最終控制權，而使用者的身分憑據也能安全地儲存在公司的伺服器上。

要和單一登入系統整合，通常會使用安全認定標記語言（*SAML*），雖然它比 OAuth 或 OpenID 標準更舊，且不太友善，但多數程式語言都具有成熟的 SAML 程式庫可供應用。


圖 9-2：需要一個臨時電子郵件位址？請找 https://www.sharklasers.com/

## 安全地重置密碼

要求每位使用者提供經過驗證的電子郵件位址，當使用者忘記密碼（無可避免）時便能派上用場，透過發送含有重設密碼的 URL 之電子郵件，並由 URL 攜帶一組新的驗證符記，當健忘的使用者開啟電子郵件並點擊 URL，伺服器驗證回傳的符記無誤，就可以開啟密碼設定頁面，讓使用者設定新密碼。

重設密碼的 URL 之有效期限應該要很短暫，而且，使用者一旦點擊此 URL，驗證符記就該立即失效，依照經驗法則，可以將驗證符記的有效期設為 30 分鐘，防止駭客濫用閒置的密碼重設 URL。若駭客入侵使用者的電子郵箱，就算找到含有密碼重設 URL 的電子郵件，也不能讓他透過這些 URL 取得受害者在網站的存取權。

```
# 從記憶體快取找出對應的連線狀態
def find_session(env, sid)
  unless sid && (session = @cache.read(cache_key(sid))❸)
    sid, session = generate_sid❶, {}
  end
  [sid, session]
end

# 將連線狀態寫入記憶體快取
def write_session(env, sid, session, options)
  key = cache_key(sid)
  if session
❷ @cache.write(key, session, expires_in: options[:expire_after])
  else
    @cache.delete(key)
  end
  sid
end
```

清單 10-1：Ruby on Rails 利用 sessionID（sid）實作伺服器端的 session 管理

❶ 處會建立 session 物件，在 ❷ 處將連線狀態寫入伺服器的記憶體快取，❸ 處會從記憶體快取讀出已被保存的連線狀態。

在 HTTP 的演進過程中，Web 伺服器嘗試很多種傳送 sessionID 的方式：將它置於 URL、放在 HTTP 標頭或寫在 HTTP 請求本文中，到目前為止，Web 開發社群覺得最常見（也最可靠）的方式是使用 session Cookie 來交換 sessionID，Web 伺服器會在 HTTP 回應的 `Set-Cookie` 標頭送出 sessionID，瀏覽器再使用 `Cookie` 標頭將相同的資訊附加在 HTTP 請求裡。

自從 Netscape 在 1995 年首次提出這種作法以來，Cookie 就已成為 HTTP 的一部分，與 HTTP 原生的身分驗證機制不同，你看得到的網站幾乎都會用到 Cookie。基於歐盟法律規定，讀者一定注意到網站會提醒它正在使用 Cookie。

幾乎各種 web 伺服器都已實作 session 管理機制，而且有一定安全度，不過，因為連線狀態要儲存在記憶體裡，系統的延展性就會受到