

## 第 14 章 安全：守护浏览器的 HTTP 功能

[日] 涩川喜规 · 详解HTTP：协议基础与Go语言实现

## 第 14 章 安全：守护浏览器的 HTTP 功能

在互联网普及之前，由可执行文件等感染计算机病毒而引起的 Crack 就是以破坏为目的的。之后，随着计算机越来越便利，攻击方法也变得多种多样。在 Windows 中添加 CD-ROM 和 USB 等的自动播放功能之后，该功能也会被用来进行攻击。互联网和浏览器能实现的功能越来越多，与此同时也出现了更多的攻击手段。破解安全漏洞，使计算机运行任意程序，虽然这种针对计算机的传统类型的攻击现在仍然存在，但随着浏览器能够执行更多任务，比如进行日常购物、使用邮件和社交网络交换个人信息等，针对计算机上的应用程序——浏览器的攻击也在不断增加。

本章将针对一些常见的安全方面的案例来介绍攻击的发生机制、防范措施和浏览器阻止这些攻击的方法。



关于 Internet Explorer 中特有的信息，我们将在附录中介绍。

### 14.1 传统类型的攻击

我们先来简单整理一下传统类型的攻击方法。注意这里的“传统”只是笔者的叫法，是指攻击不针对浏览器。传统类型的攻击的特征是访问操作系统，而非浏览器。

对计算机构成威胁的软件叫作**恶意软件**。恶意软件可根据繁殖方式和目的分为不同的类型。

繁殖方式有以下几种。首先是计算机病毒，计算机病毒会感染可执行文件等，一旦可执行文件被运行，病毒就会在其他程序中进行自我复制，不断增加。其次是蠕虫（worm），通过主动攻击网络设备和操作系统的安全漏洞来扩大感染范围。有的恶意软件会利用 USB 存储设备等的自动播放功能，或者使用乍一看很难理解的文件名（setup.exe 等）或办公软件的宏等。不管什么程序，如果不运行就没有意义，而这些恶意软件的意图就在于使用户在不经意间启动。计算机病毒有时会改写其他程序，在其他程序启动时运行。蠕虫可能并不存在宿主，在这种情况下，蠕虫会入侵操作系统的启动脚本或寄存器。

攻击方法也多种多样。以前只是以破坏为目的，比如让操作系统无法启动，或者变慢。除此之外，也存在其他攻击方法，比如随意修改操作系统的设置，设置代理服务器，以此来盗取通信内容，或者记录键盘操作，以盗取密码。有的恶意软件会设置后门，以从外部进行远程操作。还有为了破坏特定的 Web 站点的服务而同时向攻击目标发送请求的分布式拒绝服务 (DDoS) 攻击。记录键盘操作的恶意软件叫作“键盘记录器”，设置后门的恶意软件叫作“特洛伊木马”，像这样，恶意软件根据其行为方式而存在不同的类型。不过，攻击方法是慢慢被发现并命名的，名称和定义有时会根据具体情况发生变化。

虽然笔者把前面介绍的攻击方法称为传统类型的攻击，但这并不代表现在没有这种攻击了。通过浏览器插件的安全漏洞等对计算机进行攻击的情况时有发生。传统类型的攻击是对浏览器进行攻击的前期阶段，在这一阶段，恶意软件会侵入 Web 服务器，将传送的数据改写为恶意脚本，或者访问数据库以盗取用户信息。现在，以目标型邮件攻击为代表的通过发送非法程序进行的攻击具有很大的杀伤力，对 Web 服务的开发者来说，它依旧会威胁到服务的维护。

## 14.2 针对浏览器的攻击的特征

近年来，在 Web 开发者之间讨论较多的话题就是以浏览器为目标的攻击。浏览器是用于浏览的应用程序，因此不会在操作系统上进行什么操作。乍一看，与传统类型的攻击相比，针对浏览器的攻击不会造成多大的危害。然而，浏览器是连接其他服务的窗口。如果浏览器成为攻击目标，浏览器保存的用于登录各种网站的信息就会被盗，这样一来，在 Facebook 或 LINE 等外部服务中保存的私人信息以及与他人的对话等就有泄漏的风险。在进行网购或者办理网银业务的情况下，这种类型的攻击也会造成经济损失，还可能被 DDos 攻击利用。

针对浏览器的攻击的攻击目标并不只是浏览器，还有从通过浏览器浏览的服务发送来的 HTML 和 JavaScript 的漏洞。在 Web 服务中，近年来快速发展的前端新技术、横跨多个站点的账户联动、HTTPS/HTTP 的交叉等，开发者需要理解的规范越来越复杂。即使各个 Web 服务的安全功能都很强大，一旦在实现方面有所疏漏，就会出现安全漏洞。有时通过 HTTP 从正规网站发送来的文档中也会存在漏洞，但不会像计算机病毒、蠕虫程序代码、系统设置变更那样留下痕迹。对此，需要采取一些新的安全策略，比如查看通信记录来确认是否访问了可疑站点、确认下载内容等。

浏览器厂商对安全很敏感。HTTP 的规范中也考虑了很多安全方面的问题。笔者在第 2 章介绍 Referer 时也提到过，出于安全方面的考虑，有时也会对既有功能进行修改。本章接下

来将介绍常见的攻击手法和对抗攻击的 HTTP 规范。

## 会话令牌和 Cookie

Cookie 常用于在浏览器中保持会话，令牌用于确定服务器和浏览器的关系。另外，以下几项也可作为浏览器得到认证后访问用户固有内容的“通行证”使用，它们具有密钥的功能。

会话令牌

会话 Cookie

会话密钥

会话 ID

访问令牌


虽然各个术语的语感稍有不同，但在使用时意思相差不大。在本章中，笔者将 Cookie 中保存的唯一的 ID 数据（字符串）作为会话令牌，将其内容作为 Cookie 来介绍。如果会话令牌泄露了，那么相关的防御结构就会被打破，因此，可以说它起到了安全中枢的作用。

### 14.3 跨站脚本攻击

**跨站脚本攻击**（XSS）是许多攻击的源头，是 Web 服务开发者首先应该提防的攻击方法。比如，对于论坛程序等中用户输入的内容，如果不进行过滤而直接展示出来，就容易发生跨站脚本攻击。假设有一个“请输入姓名”的文本框，我们在这里输入了内容，那么就可能被其他用户看到。在 Web 服务端的程序中，在不对该输入值进行任何检查就将其直接写入 HTML 中进行输出的情况下，如果恶意脚本代替姓名被插入到 HTML 中，该恶意脚本就会直接在浏览 Web 服务的人的浏览器上运行，这种攻击方法就叫作跨站脚本攻击。

实际上，如果写入以下内容，则在每次查看该姓名时，就会显示警告对话框。

```
1 <script>alert("💩");</script>
```

 复制代码

跨站脚本攻击是所有攻击的源头，所以它是接下来介绍的攻击方法中最危险的一种。

例如，在从插入的脚本访问 Cookie 之后，传送给其他服务器的 Cookie 信息就会泄露出去。如果“登录完成”的会话令牌被盗，那么即使没有用户 ID 和密码，也可以伪造已经登录的状态。另外，跨站脚本攻击也会带来各种风险，比如登录表单被非法侵入、用户输入的信息被发送给其他服务器或钓鱼网站等。

我们可以在服务器端采用一些防卫方法，将用户输入当作持有恶意的内容，禁止将其直接输出到 HTML 中。打个比方，就是认为用户输入的内容被污染了 1，对其清洁后再使用，这种做法以前叫作“消毒” 2。一般情况下，消毒不是在输入时进行的，而是在输出之前进行的（转义）。这里所说的输出可以输出为 HTML，也可以输出为外部进程运行时的参数和数据库中运行的 SQL，后两者分别称为命令行注入（command injection）和 SQL 注入（SQL injection），它们能引起比 HTML 的跨站脚本攻击更加严重的问题。虽然不同的输出目标对应不同的转义处理，但许多模板引擎和占位符能够保证输出是安全的，使用合适的库也能减少开发者的失误。

### 14.3.1 设置 Cookie 以防止泄露

抵御跨站脚本攻击的第二道防线就是之前介绍的添加 `httpOnly` 属性的方法。在加上该属性后，Cookie 无法通过 JavaScript 访问。跨站脚本攻击的攻击者使用的就是 JavaScript。由于攻击者接触不到 JavaScript 无法访问的信息，所以能够降低 JavaScript 导致的会话令牌泄露的风险。

### 14.3.2 Content-Security-Policy 首部

使用 `Content-Security-Policy` 首部可以对网站中能够使用的功能进行详细的设定，该首部由 W3C 定义。从服务器设置网站中需要的功能后，就能够避免 JavaScript 执行预期之外的动作。

该首部可以设置的指令有 10 种以上，这些指令可大致分为 3 类。

首先是设置从 HTML 读取各种资源文件时的访问权限的指令（表 14-1）。对于超出定义范围的访问，浏览器会报错。

表 14-1 设置资源文件访问权限的指令

指令	限制对象
<code>base-uri</code>	<code>&lt;base&gt;</code> 元素中的 URI（相对路径的起点）
<code>child-src</code>	能够用作 Web Worker、 <code>&lt;frame&gt;</code> 、 <code>&lt;iframe&gt;</code> 的 URL
<code>connect-src</code>	XMLHttpRequest、WebSocket、EventSource 等通过 JavaScript 连接的通信目标
<code>font-src</code>	使用 CSS 的 <code>@font-face</code> 读取的字体
<code>img-src</code>	图像和 favicon 的读取目标
<code>manifest-src</code>	manifest 文件的读取目标
<code>media-src</code>	<code>&lt;audio&gt;</code> 和 <code>&lt;video&gt;</code> 的读取目标
<code>object-src</code>	Flash、Java Applet、插件的读取目标
<code>script-src</code>	JavaScript 的读取目标
<code>style-src</code>	可读取的样式表的读取目标

如代码清单 14-1 所示，使用分号设置各个项目。在各个项目中，表示允许读取的关键字、数据属性和 URL 都放在后面。

代码清单 14-1 使用分号设置 Content-Security-Policy

复制代码

```
1 Content-Security-Policy: img-src 'self' data: blob: filesystem;;
2                               media-src mediastream;;
3                               script-src 'self' https://store.xxxx.com
```

能够设置的数据属性如表 14-2 所示。

表 14-2 Content-Secruty-Policy 中可对资源设置的数据属性

数据属性	说明
none	禁止读取
self	指定同源
unsafe-inline	允许使用脚本的内联 <code>&lt;script&gt;</code> 标签 <code>javascript:</code> 表述、内联 <code>&lt;style&gt;</code> 。存在跨站脚本攻击的危险
unsafe-eval、 <code>new Function()</code> 、 <code>setTimeout()</code> 等	允许使用将字符串作为 JavaScript 运 存在跨站脚本攻击的危险
data:	允许 data URI
mediastream:	允许 mediastream:URI
blob:	允许 blob:URI
filesystem:	允许 filesystem:URI

大家可能不太熟悉后几个数据属性，`mediastream:` 数据属性用于 HTML5 的流。`data:` 数据属性可以将进行了 Base64 编码的图像文件的字符串设置为 `<image>` 标签的 `src` 属性值，或者在 CSS 中使用文本来填充图像数据，并进行显示。

除了针对资源文件设置的指令之外，还存在其他指令（表 14-3）。

表 14-3 针对非资源文件设置的 Content-Secruty-Policy 的指令

指令	参数类型	限制对象
referrer	no-referrer、no-referrer-when-downgrade、origin、origin-when-cross-origin、unsafe-url	修改 Referer 信息
report-uri	URL	浏览器如果检测到 CSP 违规，将其发送给指定 URL

Content-Security-Policy 是浏览器进行的检查处理。虽然客户端会显示错误，但服务器开发者并不能直接查看错误信息。通过指定错误报告的发送对象，并由客户端通知错误信息，服务器开发者也可以知道在客户端发生的问题。还存在收集 report-uri 的违规报告的 Web 服务，比如 REPORT URI（图 14-1），这种 Web 服务还可以收集下一节将介绍的 HTTP 公钥固定（HTTP Public Key Pinning, HPKP）的报告。



图 14-1 REPORT URI 官网

Content-Secruty-Policy 中还存在一些用于提高设置的安全性的指令（表 14-4）。

表 14-4 提高设置的安全性的指令

指令	参数类型
default-src	统一设置资源的访问范围。个别设置优先
sandbox	允许有弹窗和表单等。指定与 HTML 的 <iframe> 的 sandbox 属性相同的内容
upgrade-insecure-requests	将所有 HTTP 通信升级为 HTTPS

Content-Security-Policy 首部是抵御跨站脚本攻击的王牌功能，但它过于强大，甚至会影响网站的正常运行，这就好比公司的安保部门管控太严导致工作难以进行。针对这一问题，Content-Security-Policy-Report-Only 应运而生。在使用该首部执行检查的情况下，网站不会停止运行。如果停止所有的内联脚本，就会出现 Google Analytics 不能运行等情况。我们可以查看 JavaScript 控制台或者报告输出，确认网站中需要哪些功能，从而逐渐允许使用这些功能。

14.3.3 Content-Security-Policy 和 JavaScript 模板引擎

随着 JavaScript 的广泛使用，在服务器中进行的许多处理也可以在浏览器环境中进行了，其中具有代表性的就是客户端站点的 HTML 模板。

模板库有很多，这里笔者介绍一下 Hogan.js。之所以介绍它，是因为笔者了解其内部实现。Hogan.js 是由 Twitter 创建的用于 JavaScript 的模板引擎。它与 Mustache 模板引擎的语法相同，不过运行速度比较快。虽然 Hogan.js 也可以直接运行，但如果进行了模板编译，则运行速度会更快。在进行编译时，Hogan.js 会解析模板，生成内部函数调用的源代码，最后使用 new Function 动态生成函数。由于 Hogan.js 在运行时无须进行解析处理，只需展开选项和拼接字符串，所以可以快速生成字符串。也可以通过预编译提前进行编译，生成 JavaScript 的源代码。



与此相似的还有 JavaScript 框架 Vue.js。Vue.js 2.0 发布了运行时构建 (runtime-only build) 和独立构建 (standalone build) 两个版本，这两个版本的区别如下所示。

在独立构建版本中，`template` 选项中可以使用字符串来记述 HTML 模板。该模板在运行时会被转换为 JavaScript 代码，通过 `new Function` 来创建 `render()` 方法

在运行时构建版本中，不可以使用 `template` 选项。需要提前使用 `vue-loader` (用于 WebPack) 或者 `vueify` (用于 Browserify) 对 JavaScript 的源代码进行调优，以用于浏览器环境，并提前创建 `render()` 方法

当然，由于 `new Function` 是 Content-Security-Policy 的 `unsafe-eval` 指令允许执行的函数，并且会根据字符串动态生成函数，所以被禁止使用。因此，这些模板引擎成为执行 Content-Security-Policy 时的障碍。在使用这些高速的库或框架的情况下，需要进行预编译，并生成 JavaScript 代码。JavaScript 的 UI 框架 `Riot.js` 包中既包含支持 Content-Security-Policy 的版本，也包含带模板编译器的版本。

#### 14.3.4 MixedContent 的应对策略

近几年，Web 不断朝着 HTTPS 的方向发展，但在广告和外部服务提供的内容等中还混杂有 HTTP 素材。在现在的浏览器中，如果出现这种情况，就会弹出名为 Mixed Content 的错误或警告。在本书出版时，Google Chrome 计划禁止所有的 Mixed Content。由于在设置了 EV SSL 证书的情况下浏览器也会弹出警告，所以这个问题一定是服务提供者想解决的问题。

除了将所有 HTTP 都修改为 HTTPS，还有一个方法可以解决上述问题，那就是使用 Content-Security-Policy 首部的 `upgrade-insecure-requests` 指令。在使用该指令的情况下，图像等链接目标即使是以 `http://` 开始的 URL，也会按 `https://` 来处理。

**即使记述为 HTTP，也会按 HTTPS 来处理 (1)**

Content-Security-Policy: upgrade-insecure-requests

也可以在 HTML 的元标签中记述 `Content-Security-Policy`。

### 即使记述为 HTTP，也会按 HTTPS 来处理 (2)

```
<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">
```

除此之外，我们还可以使用完全阻塞的方法。

### 将 Mixed Content 作为错误处理

```
Content-Security-Policy: block-all-mixed-content
```

Google 网站上记录了更加详细的信息。

## 14.3.5 CORS

**CORS** (Cross-Origin Resource Sharing, 跨域资源共享) 由 W3C 实现标准化。所谓资源共享，就是使用 `XMLHttpRequest` 和 `Fetch API` 进行访问。虽然使用 `Content-Security-Policy` 首部的 `connect-src` 指令可以对使用这些 API 的访问进行控制，但 CORS 的主要用途是严格执行外部服务的访问控制。

用一句话来说，CORS 就是客户端访问服务器之前的权限确认协议。目前，CORS 更多地出现在“因为 CORS 而无法提供 API”这类话题中，但它原本是安全方面的功能。不过，CORS 守护的对象并不是客户端，而是 API 服务器。Web API 会提供有价值的信息，CORS 则会防止未经允许的网站“免费使用”这些有价值的信息。虽然 CORS 看起来很麻烦，让人很难理解其用途，但如果将这些前提记在脑子里，就很容易理解了。

另外，这些 CORS 的流程并未出现在 JavaScript 的源代码中，而是在如代码清单 14-2 所示的 `fetch()` 函数等的后台悄悄执行。

### 代码清单 14-2 悄悄执行 CORS 流程

```
1 fetch('https://api.external.com, {
```

 复制代码

```
2     method: 'PATCH',
3     mode: 'cors',
4  }).then(function (response) {
5     // 请求通过时调用的回调函数
6  });
```

能否进行通信是根据请求的条件或者服务器的响应来判断的。从客户端的角度来看，省略了详细的错误处理的流程如图 14-2 所示。通信结果有“进行通信”和“失败”两种。失败时会显示 JavaScript 的 `NetworkError` 异常等。

流程大致分为 simple cross-origin request 和带有预检请求的 actual request 两种。预检请求是在实际通信之前为了确认权限而发送的请求。simple cross-origin request 成立的条件如下所示。

HTTP 请求的方法是简单方法 (`GET`、`POST`、`HEAD` 中的一个)

首部只有简单首部 (只包含 `Accept`、`Accept-Language`、`Content-Language`、`Content-Type`)

在包含 `Content-Type` 的情况下，其值为 `application/x-www-form-urlencoded`、`multipart/form-data`、`text-plain` 中的一个

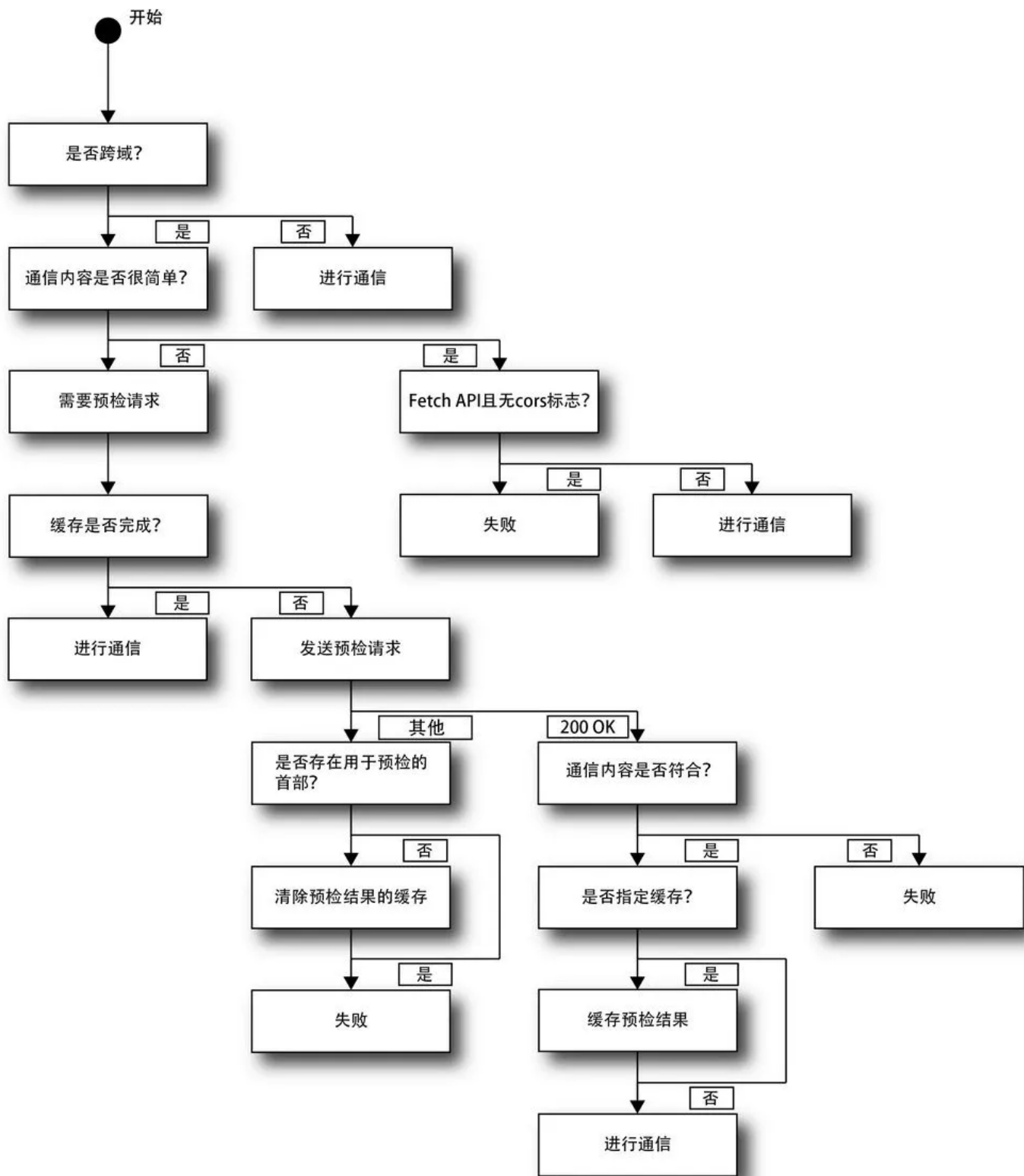


图 14-2 判断是否进行通信的流程（省略错误处理）

在不符合这些条件的情况下，必须执行预检请求。在执行预检请求时，客户端附加以下首部，使用 `OPTIONS` 方法发送请求。

## Access-Control-Request-Method 请求首部

列举希望获得许可的通信的方法，并用逗号分隔。

### **Access-Control-Request-Headers** 请求首部

列举希望获得许可的首部，并用逗号分隔。

### **Origin** 响应首部

指定通信源的 Web 页面的域名。

前面的流程图中没有涉及 Cookie 的处理，因为在默认的跨域通信中并不发送和接收 Cookie。客户端使用设置好的 Fetch API (`credentials: 'include '`) 或者 XMLHttpRequest (`xhr.withCredential = true`)，仅在服务器允许的情况下发送请求。

服务器接收这些通信，使用以下首部将服务器允许的通信内容通知给浏览器。在不允许通信的情况下，则不附加各个首部，或者以 `401 Forbidden` 的形式返回状态码。

### **Access-Control-Allow-Origin** 响应首部

允许通信的域名。在不使用 Cookie 的情况下，使用通配符 (\*) 来统一允许使用所有的域，否则明确记述请求的域名。

### **Access-Control-Allow-Method** 响应首部

针对对象 URL 允许使用的域名列表。不需要预检请求的简单方法有时会省略该首部。

### **Access-Control-Allow-Headers** 响应首部

针对对象 URL 允许使用的首部名称列表。不需要预检请求的简单方法有时会省略该首部。

### **Access-Control-Allow-Credentials** 响应首部

在允许服务器接收 Cookie 等资格信息时附加的首部。可以设置的值只有 `true`。

### **Access-Control-Expose-Headers** 响应首部

指定在服务器返回的响应首部中，有哪些首部可以作为响应的一部分暴露给客户端。

Web 浏览器基于这些信息与实际要发送的内容进行比较，然后判断是否可以进行通信。如果要发送的内容都是得到允许的，就开始通信，否则返回错误。

并非每次请求之前都要发送预检请求。使用以下首部，可以通过将通信内容缓存一定时间来省略通信。

与使用了 `Cache-Control` 的缓存一样，服务器通知客户端可以缓存的秒数。

## 14.4 中间人攻击

**中间人攻击**（Man-in-the-Middle Attack，MITM 攻击）是代理服务器中继通信时通信内容被盗取而造成的信息泄露问题。目前，通过免费 Wi-Fi 来使用互联网的情况在国内外变得越来越常见，如果其中设置了恶意访问点，那么连接互联网的线路中的数据包就可能被偷窥，从而导致通信内容泄露。特别是一旦泄漏了用户 ID、密码或会话令牌等，其他信息也会发生泄露，甚至连内容也会被改写。2015 年 3 月发生了针对 GitHub 的 DDoS 攻击（Distributed Denial of Service Attack，分布式拒绝服务攻击）。当时，普通用户所读取的 JavaScript 被篡改，其中嵌入了每两秒访问一次 GitHub API 的代码，最终导致服务因大量访问而无法正常运行。除此之外，多次执行浏览器发送的收费处理，给用户造成经济损失等情况也时有发生。

如果直接通过 HTTP 来使用网站的登录表单进行登录，用户名和密码就可能会被盜。即使不进行登录，如果会话令牌被盜，受保护的信息也可以被访问。只有使用 HTTPS（TLS）才能避免中间人攻击。TLS 是即使在通信线路不可靠的状态下也能保证通信安全的一种结构，可以防止第三者监听通信内容、改写通信、发送客户端或者服务器预期之外的请求等。不过，正如第 4 章介绍的那样，TLS 保护的只是通信线路，因此还需要注意针对服务器的攻击和针对浏览器的跨站脚本攻击。

会话令牌保存为 Cookie，如果条件（URL）符合，就在发送请求时加上 Cookie。正如在介绍跨站脚本攻击时提到的那样，通过在服务器附加 Cookie 时指定 `secure`，可以只有在有 TLS 保护通信线路的情况下发送请求，从而进一步抵御中间人攻击。

## HSTS

RFC 6797 中定义的 **HSTS**（HTTP Strict Transport Security，HTTP 严格传输安全）是抵御中间人攻击的一种 HTTP 结构。该结构用于让服务器通知“今后连接时请用 HTTPS 连接”的消息。服务器在希望使用 HTTPS 连接时，会将以下首部附加到响应首部中。如果存在 `includeSubDomains`，则会告诉浏览器，子域也是跳转对象。当然，在返回该首部时，也可以同时进行重定向。

```
1 Strict-Transport-Security: max-age=31536000;includeSubDomains
```

浏览器内部持有发送该首部的 URL 数据库。当浏览器访问指定的站点时，会自动连接 HTTPS 站点。该指示只能在有效期限内使用。上面的代码表示浏览器在 31 536 000 秒（即 1 年）的时间内会自动连接 HTTPS 进行传输。

HSTS 结构存在缺陷。在第一次连接时，服务器要求使用 HTTPS 连接的指示还未到达，因此第一次连接是 HTTP 通信。在这种状态下，如果存在恶意的代理改写了重定向目标的 URL，浏览器就可能被引诱到钓鱼网站。TLS 中存在签名，所以浏览器能检测到改写，但在第一次连接使用 HTTP 的情况下，浏览器无法检测到该攻击。

为了解决该问题，还出现了新的防护措施。Google Chrome 进行了信息收集，以从一开始就设置好 URL 数据库。通过事先从网站进行申请，只需下载最新的浏览器，就可以在第一次连接时就使用 HTTPS。这里申请的信息会用在 Chrome、Firefox、Safari、Internet Explorer 11、Edge 和 Opera 等各种浏览器中。虽然 RFC 中没有写明，不过在申请信息时，会将 `preload` 指令附加到 `Strict-Transport-Security` 首部中。

针对 HTTPS 的中间人攻击中存在一个问题，如果中间人持有的证书被承认了，便可以合法地打破 HTTPS 通信的秘密。认证机构设置错误或者被攻击而导致一些不合适的证书被发行的事件时有发生。2016 年 9 月，沃通（WoSign）被发现可以通过验证子域名控制权的方式获得签发主域名的数字证书，这样一来，用户就会拥有超过允许范围的权限，该问题无法用普通方法解决。



HTTPS 安全通信的第一步就是使用正确的密钥。如果可以任意操作操作系统的设置，信任恶意证书，那么 HTTPS 通信的前提也就不存在了。作为对策，2015 年 4 月 RFC 7469 中定义了 HPKP（HTTP Public Key Pinning，HTTP 公钥固定）这一安全功能。该功能通过将证书中内置的公钥预先保存在浏览器中来检查证书是否被篡改。不过，一旦 HPKP 部署错误，就会出现无法访问正确网站的问题，最早实现该功能的 Chrome 也已经不支持该功能了，现在只有 Firefox 支持它。Chrome 实现了使用新的

Expect-CT 首部来确认证书的功能，但除了 Chrome 及其派生的浏览器，其他浏览器还不支持该功能。

## 14.5 会话劫持

**会话劫持** (session hijacking) 是通过盗取 Web 服务的会话令牌来登录网站的一种攻击方式。在普通的 Web 服务中，第一次输入用户名和密码之后，会话令牌会作为 Cookie 发送给浏览器。浏览器持有该 Cookie，在第二次及之后的访问中就不再需要用户名和密码了。如果该会话令牌被盗，就能够以登录的状态来访问网站，这就相当于泄露了用户名和密码。这样一来，就会出现通过不正当的手段在购物网站上随便下单等情况。

盗取 Cookie 这一行为的基础是跨站脚本攻击和中间人攻击。下述方法可以抵御这些攻击，有效避免会话劫持。

HTTPS 化

Set-Cookie: httpOnly、secure

### 14.5.1 旧的会话管理和会话固定攻击

**会话管理**是指识别用户的客户端。人们很早之前就采取了各种各样的方法来实现会话管理，常用的方法有使用客户端持有的固有 ID。

功能手机中经常使用个体识别编号这一终端固有的 ID 进行识别。比如 DoCoMo 就是通过 `guid=on` 来获取个体识别编号的。与之类似的还有在 iOS 7 之前的版本中可以使用的终端识别编号 UDID、从 iOS 8 开始不再使用的、利用了 MAC 地址的终端识别方法。MAC 地址是通信接口标识符，它是唯一的。所谓“利用了 MAC 地址的终端识别方法”，具体来说就是使用操作系统的 API 进行读取，并像会话令牌一样将其附加到请求中。因为从 iOS 8 开始 MAC 地址可以随机生成，所以就不能用这种方法来识别客户端了。

个人信息不止包括容貌和姓名等，如果连接信息的结果与特定的某个人有关，那么该结果也属于个人信息。不管是在 A 网站，还是 B 网站，只要使用的终端相同，终端识别编号就是同一个。假设 A 网站持有终端识别信息和姓名的数据库，B 网站持有终端识别信息和住址的



数据库，C 网站持有终端识别信息和照片的数据库，那么只要获取这三个数据库，就能通过终端识别信息知道某个人的容貌、住址和姓名。

除了隐私方面的问题，还存在安全方面的问题。在功能手机中，终端识别信息会写入用户代理的字符串或首部中，而读取或冒用该信息是一件非常简单的事情。另外，用户无法随意修改该信息，这对攻击者来说非常有利。当时的功能手机中没有 Wi-Fi 连接，而且全球 IP 地址的范围是确定的，因此，我们也可以通过 IP 过滤来防御攻击，不过现在最好使用其他方法。

其中一个方法就是将会话令牌填充到 URL 中。这也是为了让 Cookie 无法在功能手机中使用而采取的方法。会话令牌保存在名为 `JSESSIONID` 或者 `PHPSESSIONID` 的键中。URL 是作为 Referer 发送的信息，如果用户将 URL 复制并粘贴到 SNS 等上，会话令牌就会泄露。如果（在填充到 URL 时）服务器实现较差，登录前后会话令牌不会发生变化，就会发生问题。

如果经由包含了攻击者创建的任意会话令牌的 URL 进行访问，创建的字符串就会升级为正式的会话令牌，这样一来，知道该会话令牌的攻击者也能够自由访问。这就是**会话固定攻击**。

## 14.5.2 Cookie 注入

在无状态的 HTTP 中，保存浏览器状态的 Cookie 在给我们带来方便的同时，也常常有被攻击的危险。一些攻击方法就是专门针对 Cookie 的，Cookie 注入就是其中之一。

**Cookie 注入**是反向获取 Cookie 规范的方法，可以迂回执行 HTTPS 连接。刚发布时的攻击手法是用其他非 HTTP 的子域覆盖被 HTTPS 隐藏的域的 Cookie，或者设置 URL 的详细的 Cookie，使原来通过 HTTPS 指定的域的 Cookie 无效，为会话固定攻击做铺垫。上一节介绍的功能手机的会话固定攻击就是因为无法使用浏览器的 Cookie 而时常发生。在通过其他环境的浏览器使用 Cookie 的情况下，如果在登录前后服务器未重新分配会话 ID，通过这些方法就能够从外部设置会话令牌，因此需要多加注意。

2017 年 3 月，Chrome 和 Firefox 针对 Cookie 注入采取了应对措施，现在无法再通过子域设置 Cookie，而且即使在同一域中，也不可以通过 HTTP 来覆盖带 `secure` 的 Cookie。虽然已经有了用于标准化的草案，但其他浏览器还未对其进行实现<sup>3</sup>。

## 14.6 跨站请求伪造

正如前面介绍的那样，HTTP 是无状态的。即使浏览器使用 GET 获取表单的请求，以及将值存储到该表单中后以 POST 方法发送的请求是从其他国家发送过来的，或者只发送了 POST，结果也是一样的。假设在图像标签中写入了 URL，浏览器要访问该 URL。如果有一个使用 GET 来修改数据的 API4，那么“恶意的图像标签”就会导致该 API 被恶意使用。

**跨站请求伪造**（Cross Site Request Forgery, CSRF）能够从无关的页面或网站发送非本人意愿的服务器请求。实际上，发送请求的并不是攻击者，而是受害的用户，Web 浏览器会向被诱导的页面发放 Cookie，因此用户会保持登录状态。这样一来，攻击者就能够以受害用户的权限来执行任意操作。这种强制执行受害者预期之外的操作的攻击就是跨站请求伪造。

### 14.6.1 应对跨站请求伪造的令牌

给 HTTP 的无状态性添加限制是防范跨站请求伪造的常用方法，具体来说就是在设置表单时，在隐藏的字段（type 为 hidden 的字段）中填入随机生成的令牌，使接收 POST 请求的服务器拒绝所有未包含正确令牌的请求。Web 应用程序框架中也应该包含用来生成和验证令牌的中间件。

浏览器并不需要关心应对跨站请求伪造的令牌，只需直接返回发来的表单，因此不需要拥有什么特殊的功能。

另外，提起每个用户固有的值，我们会想到会话令牌，但会话令牌不可以作为应对跨站请求伪造的令牌使用。在 Cookie 中可以使用 httpOnly 进行保护，但应对跨站请求伪造的令牌的实现方法中经常会用到表单的隐藏字段，因此 Cookie 的安全性无法得到保护，可以通过 JavaScript 轻松被访问。mala 也提到 HTML 文件中存在许多泄露方法。在使用考虑到便捷性而添加的网页剪裁工具时，HTML 文件也会被发送到外部服务器。如果会话令牌和应对跨站请求伪造的令牌相同，会话劫持的危险就会进一步增加。

与应对跨站脚本攻击的方法一样，直接使用 Web 应用程序框架中提供的结构是最安全的，它会自动生成与会话令牌不同的令牌。

### 14.6.2 SameSite 属性

在 Cookie 中加上 `SameSite` 属性后，只要发送请求时的页面不在同一个站点中，就不用再发送 Cookie 了，这样就无法从无关的站点发送请求，从而能够防范跨站请求伪造。详细内容请参考第 2 章中对 Cookie 的介绍。

## 14.7 点击劫持

**点击劫持** (Click Jacking) 的事例有两种，它们都使用 IFRAME。

一种是将 Twitter 等常用的网站透明化，然后重叠到恶意页面上。恶意页面是显示给用户的。例如，将“点击查看更多信息”等诱导点击的按钮与常用的网站的退出按钮、社交网站的分享按钮等放在一起，让用户在无意间执行这些操作，这样就对实际的网站直接执行了与跨站请求伪造相同的操作。

另一种正好相反，将常用的网站显示在下面，在上面使用透明的层来显示恶意页面，并冒充正规页面，使用户对其进行操作。

由于只是显示在 IFRAME 的内部，在网站看来，用户的操作与平时一样，因此我们无法采取与跨站请求伪造类似的对策。为了应对点击劫持，浏览器需要防止页面在 IFRAME 内使用。目前，在除了 Internet Explorer 之外的其他浏览器中，可以由正规网站使用 `Content-Security-Policy` 首部的 `frame-ancestors`，来防止浏览器被恶意使用。

浏览器通过查看该首部而拒绝某些显示，以此来保护用户。

```
Content-Security-Policy: frame-ancestors 'none'
```

拒绝在框架内使用。

```
Content-Security-Policy: frame-ancestors 'self'
```

拒绝在同一个 URL 之外的框架内使用。



### 同时确认所有安全相关的首部的 Web 服务

在 Security Headers (图 14-3) 中输入服务的 URL，就可以同时确认是否附加了安全相关的首部等。

有的语言和框架提供了向 Web 服务统一附加安全首部的中间件，如下所示。如果检查结果存在问题，我们就可以考虑导入中间件。

Go: Secure

Node.js: Helmet

Django: Security middleware 等

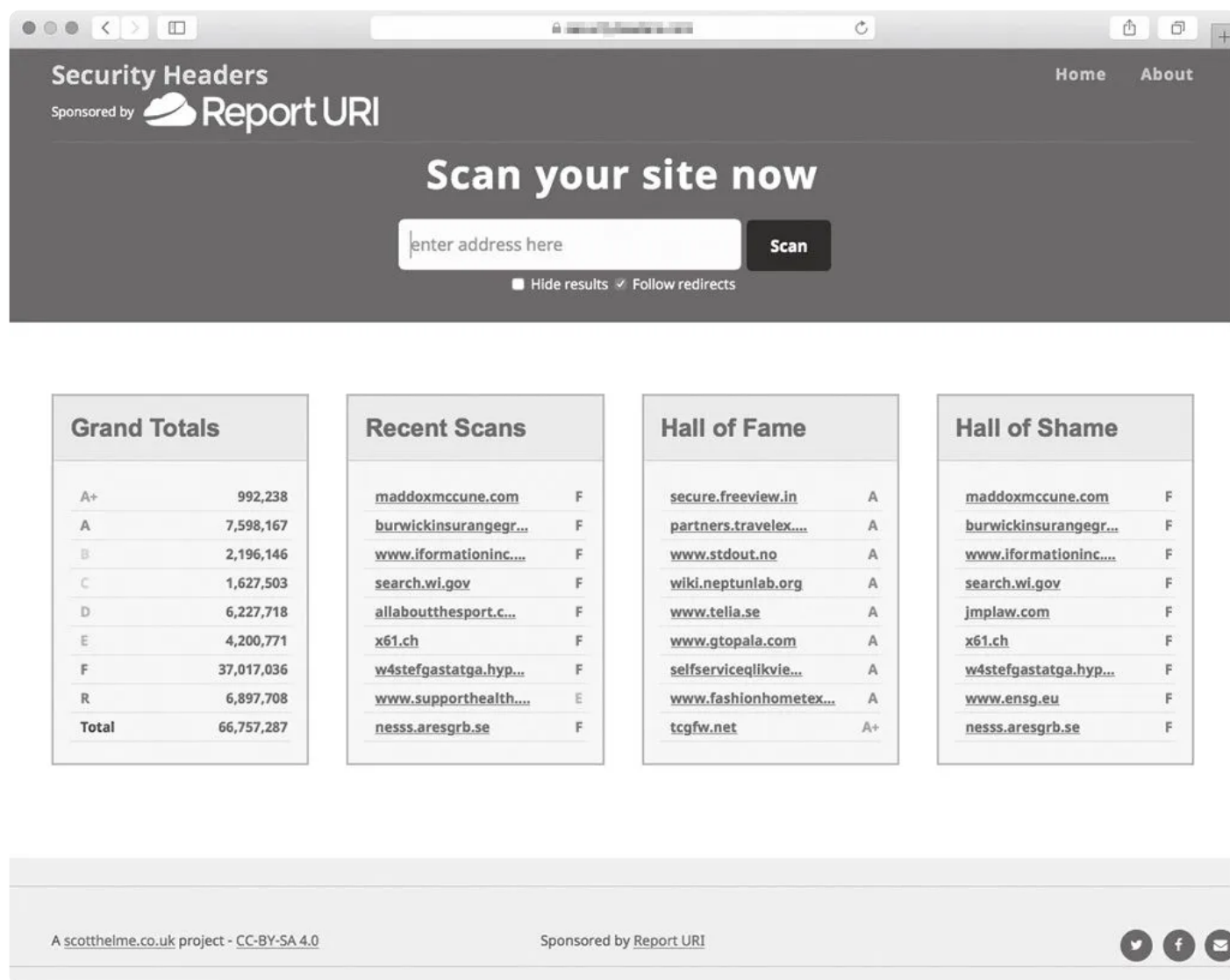


图 14-3 Security Headers 官网

## 14.8 列表型账户入侵

近年来，**列表型账户入侵**的问题比较严重。当脆弱的 Web 服务器被非法侵入，用户的登录 ID 和使用明文保存的密码被泄露时，使用相同邮箱地址和密码的账号就不再安全了。

前面介绍的攻击手法都以某种形式介入客户端和服务端之间的通信，而列表型账户入侵则与实际通信无关。前面的事例中也介绍了抵御攻击的浏览器功能（首部），但如果浏览器不参与通

信，我们就无法使用浏览器功能来抵御攻击了。

对网站运营者来说，列表型账户入侵是一个棘手的问题。即使原因在于其他网站泄露了密码，为了坦诚地对待用户，遭受攻击的服务也会主动公开。诚然，这样的做法是对用户负责的，但公司的品牌会因此受到影响。


接下来，笔者将介绍服务端的保护方法。除此之外，实际上还存在不采取这些措施（公司内不保存密码）的方法。例如，使用 Google、Facebook 和 LINE 等 ID 平台的方法，以及使用 Auth0 或 AWS Cognito 等认证 API 的方法。自身不持有用户信息的做法风险最低，也能够轻松使用最新的安全保护策略。用户也减少了注册的麻烦，服务用起来也更加方便 5。下面，笔者先来介绍使用外部服务的方法。

### 14.8.1 密码的保存：不保存明文密码

列表型攻击的第一步是，趁操作系统设置不完善时从外部入侵，下载会员信息。防止用户 ID 和邮箱地址、密码泄露是管理用户 ID 的服务运营商的责任和义务。我们经常能看到使用明文保存的密码遭到泄露这样的新闻。即使密码泄露，只要不是明文密码，就可以防止列表型攻击。


持有密码的系统都使用了第 4 章介绍的散列函数。为了便于说明，本节使用 MD5。另外，由于 MD5 不具有强抗冲突性，所以它不可以用来保存密码。在实际进行实现时，请使用下一节介绍的 bcrypt 等更可靠的散列算法。

假设用户密码是“mizuho-bank”。我们对密码进行散列处理，并将其保存到数据库中。在用户登录时，通过执行同样的处理来判断散列值是否相同，这样一来，就算不保存明文密码，也可以进行确认。

 复制代码

```
1 $ echo mizuho-bank | md5
2      86b817e72ada5353c28b3029a50374a3
```

不过，脆弱的密码（如常用密码排行榜中的 password 等）即使经过了散列处理，也能通过散列值反推出来。反推的计算量较大，但如果使用计算完成的散列值对照表——彩虹表，密码也可能被瞬间破解。鉴于此，后来又出现了一种通过添加被称为盐的字符串，让散列值变为完全不同的值的方法。例如，添加 "-tako" 字符串后，结果会发生改变，从而能够防止散列值泄露。

 复制代码

```
1 $ echo password | md5
2     286755fad04869ca523320acce0dc6a4
3
4 $ echo password-tako | md5
5     d0161bce05fffb6562ebaa3116c6c80e
```

另外，从理论上说，不保存明文密码，就表示当用户找回密码时，Web 服务无法将密码显示在页面上，或者通过邮件发送密码，而只能提供重置密码的功能。另外，在修改密码时，Web 服务无法检测到当前密码与之前的密码是否有 3 个字符以上的不同。如果读者收到产品负责人或系统开发委托人的这种开发请求，必须坚决拒绝。

## 14.8.2 保存密码时使用的各种散列函数

保存密码时可以使用的算法有如下几种类型。

### **bcrypt**

1999 年开发的可在 C、C++、C#、Go、Java、JavaScript、Elixir、Perl、PHP、Python 和 Ruby 等语言中使用的经典算法。持有循环次数参数，如果参数指定为 12，则循环执行  $2^{12} = 4096$  次加密处理。

### **PBKDF2**

Wi-Fi 的 WPA2 中也可以使用的散列算法。与 bcrypt 相同，也可以设置循环次数。

### **Argon2**

2015 年 7 月获得 Password Hashing Competition 冠军的算法。该算法使得在 GPU 上进行并行计算非常困难，运行时间、可用内存大小和并行处理数都可以通过参数进行设置。

### **scrypt**

2009 年开发的算法，该算法在 RFC 7912 中实现标准化。其内存占用量较大，使用专用硬件也很难破解。该算法用于很多加密货币中，使用 GPU 可实现高速计算。这就导致全球 GPU 的需求量剧增，GPU 和内存等设备的价格上涨。

Web 开发的相关图书中会介绍到，保存密码时应该使用加盐散列值，不可以明文保存，而最近的 Web 框架会默认带有密码的散列化结构。PHP 可使用内置的 `bcrypt`。Ruby on Rails 教程中介绍了使用 `bcrypt` 对密码进行加密的方法。Django 默认使用了 SHA256 的 PBKDF2 算法有效，也可选用 Argon2 或 `bcrypt`。Java 的 `SpringSecurity` 也支持 PBKDF2、`bcrypt` 和 `scrypt` 等。Go 语言的标准库提供了各种算法。

`bcrypt` 是经典算法，从 1999 年问世到本书执笔时，已经过了 20 年。`bcrypt` 持有循环次数参数，当该参数值增大时，处理时间也会变长，这与其他散列函数不同。如果循环次数参数为 12，则循环 2 的 12 次方，即 4096 次。大量的运算可以在一定程度上抵御盗取密码的暴力攻击。

### 14.8.3 密码的日志掩码化

近年来，为了跟踪问题、改善用户操作，我们通常会收集访问日志等各种日志信息，而这也引发了几起在日志中输出了敏感信息的安全事故。针对这种情况，需要在输出日志时进行掩码化，将密码信息隐藏起来（变为空字符串或伪字符串）。

另外，敏感信息并不仅限于密码，需要以明文状态提供的数据（如地址、信用卡号等）也要注意。从个人信息的角度来看，我们需要限制从公司内部访问个人信息的手段，严格管理可访问个人信息的员工的权限。不过，在通过调试日志查看信息的情况下，这些控制可能会无效，因此，我们需要根据具体情况来进行处理，比如使用机器学习分析数据，在查看住址等的情况下不关联用户信息，仅通过散列化等方法来判断数据是否相同。

### 14.8.4 多因素身份验证

最近，为了应对列表型攻击，Web 服务开始广泛使用**多因素身份验证**（Multi Factor Authentication, MFA）。

根据因素数量的不同，有两步验证（2 Step Verification）、双因素身份验证（2 Factor Authentication, 2FA）和多因素身份验证。这里我们来介绍一下最灵活的多因素身份验证。

多因素身份验证由用户记忆的因素、用户持有的因素、用户与生俱来的因素组合而成<sup>6</sup>。密码是用户记忆的因素，软件和硬件令牌等是用户持有的因素，指纹、虹膜验证和人脸验证是用户与生俱来的因素。例如，密码和密保问题都是用户记忆的因素，因此，这两个因素不构成多因素身份验证。

### 14.8.5 TOTP 算法

在多因素身份验证中，常用的设备验证算法是 **TOTP 算法**（Time-based One-time Password Algorithm）。该算法定义在 RFC 6238 中。想必很多人见过随机变化的 6 位数值，这就是基于该算法生成的。

从服务的角度来看，引入 TOTP 算法相当于向系统中添加了第二个密码。不过，用户不可以任意决定、修改该密码（准确来讲是密钥）。首先，生成足够随机的复杂位串，位串的长度根据算法发生变化。使用 HMAC-SHA-1 时生成 20 字节的随机数的密钥；使用 HMAC-SHA-256 时生成 32 字节的随机数密钥；使用 HMAC-SHA-512 时生成 64 字节的随机数密钥。密钥中包含发行者的名称、账户名等，类似于 URL。方案名是 `otpauth`。

在验证时，TOTP 的移动应用程序根据密钥和当前时间生成数值串。常用的是 6 位数值，每 30 秒生成一次新的字符串。用户将该移动应用程序生成的 6 位数值发送给服务。服务端使用自己的密钥和当前时间进行计算，如果计算出来的结果与该 6 位数值相同，则判断用户持有正确的密钥。

需要注意的是，密钥必须严格管理。在用户扫描服务端显示的 QR 码注册移动应用程序之后，如果废弃该 QR 码，那么就不可以再使用该密钥注册应用程序了。因此，提供 TOTP 的各个服务都没有再次告知之前的密钥的功能（重新显示 QR 码），这些服务仅提供再次生成密钥的选项。

Go 语言中有很多用于 TOTP 验证的库，下面来介绍一下 GitHub 上关注最多的使用了 `pquerna/otp` 包的实现方法。



```
1 package main
2
3 import (
4     "image/png"
5     "log"
6     "os"
7     "time"
8
9     "github.com/pquerna/otp/totp"
10 )
11
12 func main() {
13     // 在用户注册时生成密钥
14     // 生成 PNG 文件的 QR 码, 并注册到用户的双因素身份验证应用程序中
15     // 将该密钥保存到数据库中
16     key, _ := totp.Generate(totp.GenerateOpts{
17         Issuer:      "xxxx.com",
18         AccountName: "alice@xxxx.com",
19     })
20     f, _ := os.Create("qr.png")
21     img, _ := key.Image(200, 200)
22     png.Encode(f, img)
23
24     // 在用户登录时生成校验码, 通知用户
25     // 由验证应用程序执行, Web 应用程序中不改写该校验码
26     code, _ := totp.GenerateCode(key.Secret(), time.Now())
27     log.Println("code:", code)
28
29     // 基于生成的密钥来验证用户输入的密码
30     // 如果正确, 则登录成功
31     valid := totp.Validate(code, key.Secret())
32     if valid {
33         log.Println("success!")
34     }
35 }
```

虽然这些代码都写在了一个函数中, 但是执行 TOTP 登录的逻辑的时机有 3 个。首先是在用户注册时生成密钥, 然后是在用户登录时生成 6 位校验码。通常这是由用户在智能手机等中安装的应用程序执行的, 因此无法在其他相同逻辑的服务上执行。最后是验证用户输入的 6 位校验码。通过该验证, 再加上用户 ID 和密码, 就可以实现多因素身份验证了。

## 14.8.6 WebAuthn

FIDO (Fast IDentity Online, 线上快速身份验证) 联盟是为线上快速身份验证制定标准的机构。FIDO 发布了 U2F (FIDO Universal 2nd Factor, 通用第二因素) 等强化验证的解决方案。WebAuthn (Web Authentication, Web 身份验证) 使得浏览器可以使用 U2F 设备 (如 Yubikey 等)、智能手机等的指纹验证、Windows Hello 等硬件和操作系统功能。2019 年 3 月, WebAuthn 成为 W3C 标准, 在笔者编写本书时, Chrome、Edge、Firefox 和 Safari 已经对 WebAuthn 提供了支持。

通过支持 WebAuthn, 可以将验证时的用户操作控制在最小限度, 同时能够保证多因素身份验证的安全性。

除了浏览器的 API 之外, WebAuthn 还定义了服务器的动作。浏览器的 API 只有两个, 一个用于用户注册, 一个用于登录。

```
navigator.credentials.create()  
  
navigator.credentials.get()
```

验证采用挑战 / 响应 (Challenge/Response) 方式。客户端对服务器创建的 “挑战” 进行加工, 并发送给服务器进行验证。WebAuthn 与 TOTP 相似, 但返回的是服务器生成的 “挑战”, 而不是时间。与使用密码的方式不同, 由于每次验证时的 “响应” 都不一样, 所以更能抵御攻击。上面两个 API 都会接收服务器生成的 “挑战”, 并将结果发送给服务器进行确认。第 5 章介绍的 OAuth2 的 PKCE 是客户端创建 “挑战”, 而 WebAuthn 则与其相反。

服务器和客户端的示例代码非常长, 本书就不详细介绍了, WebAuthn 的网站本身就是示例代码, 这些代码发布在了 DuoLabs 的 GitHub 上。另外, Go、Java 和 Python 等各种编程语言可以使用的服务器库等也都公布在了 OSS 上。

### 14.8.7 通知用户别处的登录

为了减轻被攻击后的损失, 许多公司采用了通知登录的方法。通过向用户的邮箱或者信赖的设备发送通知, 能够让用户注意到其他地方的登录。还有一种使用了 Geo-Location 的方法。笔者在第 6 章中也介绍过, 可以根据使用 IP 地址获取的位置信息来实现过滤。如果在生

活场所之外的地方出现了访问，可以暂停访问，再次进行验证。不过，检查得越精细，就需要越精确的位置信息。如果攻击者与被攻击者的位置很近，就无法抵御攻击了。有时也会不使用 Geo-Location，而是通知从浏览器的用户代理那里获取的信息，比如“从 Windows 设备登录”“从 Mac 登录”等。

有时位置信息也会成为多因素身份验证的因素，但笔者从未看到过位置信息被作为用户验证的因素使用。另外，在使用浏览器的 JavaScript 获取并发送位置信息的情况下，如果客户端被非法入侵，攻击者就可以对位置进行伪装。使用 VPN 从其他服务器访问网络，还可以伪装 IP 地址。由于位置信息的准确性很低，所以很难用于验证。

图 14-4 是 Kyash 发送的邮件，这种简单的邮件对及早发现列表型攻击有很大的帮助。



图 14-4 Kyash 向用户注册的邮箱地址发送的邮件 <sup>7</sup>

## 14.9 注入存在漏洞的代码

现在，我们基本上不会只使用标准库进行软件开发。得益于各种开源库，我们可以在短时间内开发高质量的应用程序。而与此同时，对库进行的攻击也成为不容忽视的问题。

广受欢迎的 `event-stream` 库就曾经发生过被植入恶意代码的严重事件。恶意开发者从维护人员那里接手发布权限，在库中植入了盗取比特币的代码。虽然如此严重的恶意事件很少见，不过存在将自己的库伪装成受欢迎的库的情况<sup>8</sup>，还有不小心在库中注入漏洞的情况。

针对库和工具的漏洞，可以利用 CVE、JVN 等数据库列表和 VulnDB 等收费的数据库，来了解存在的漏洞。2019 年，仅 CVE 就记录了 2 万个漏洞。每天都有漏洞被检测出来，相应的安全补丁也每天都在发布。漏洞数量之多，使得检测漏洞、获取并应用补丁成为一项非常庞大的工作。IBM 每年发布的网络安全报告显示，恶意利用已修复的漏洞的攻击还有很多。

为此，许多检查漏洞的工具应运而生，比如 Vuls、Trivy 等 OSS 工具和 Snyk 等免费工具。Node.js 的包管理工具 npm 中也新增了漏洞检测功能，GitHub 也提供了同样的功能。另外，Vuls 中的 FutureVuls 也提供了管理漏洞的界面。这些工具是安装在操作系统中的包或程序开发者创建的应用程序的依赖库等，各工具的漏洞检测对象有所不同，检测精度也不一样。我们需要使用这些工具及早修复漏洞，确保应用程序和服务的安全。

## 14.10 面向 Web 应用程序的安全指南

到这里为止，笔者介绍了各种安全方面的浏览器功能，如下所示。

`Content-Security-Policy` 首部

`Strict-Transport-Security` 首部

`Public-Key-Pins` 首部

`Set-Cookie` 首部 (见第 2 章)

应对跨站请求伪造的令牌

多因素身份验证

`GeoIP` (见第 5 章)

`X-Content-Type-Options` 首部 (见第 1 章)

`X-Frame-Options` 首部 (用于 Internet Explorer, 见附录 A.3)

大家可以参考一下 OWASP 发布的安全方面的备忘录，IPA（信息处理推进机构）网站上也发布了相应的指南。通过本章大致了解存在什么样的攻击、浏览器是如何被欺骗的之后，我们就能轻松理解这些资料了。另外，安全相关的图书也有很多。

## 14.11 Web 广告和安全

电视广告以“收看相同节目的观众喜好也基本相同”这一认知为前提来发布相同的广告。

Web 广告则更进一步，即使是同一个网站，也可以针对每个用户发布不同的广告。要想高效地发布吸引用户的广告，广告从业人员就需要获取与用户兴趣有关的信息，这一点非常关键。

不过，兴趣这种无形的信息是很难获得的，因此广告从业人员会通过获取用户访问的 URL 列表来推测用户的爱好。但是，要想一并获取用户访问的网站的信息是不可能的，于是广告从业人员通常采取的方法是，给用户分配 ID，对于所跟踪的各个网站，分别记录用户访问的信息，将各个零散的信息连接起来，来还原用户的浏览记录。

现在使用的统计方式大致分为两种。在使用基于 Cookie 的统计工具的情况下，由于会将固有 ID 提供给用户，所以我们能够获取从点击开始 3 个月以上的转化结果。

另一种方式叫作 Finger Print。根据终端浏览器的版本信息、IP 地址、地域信息和机型信息等，使用各种方法对用户进行详细分类。与基于 Cookie 的统计工具不同，这种方式无法确定具体的某个用户。它的优点是无须在终端设置 Cookie 之类的内容，Finger Print 信息可以随时根据容易观测的数据生成。不过如果间隔了两周左右的时间，用于生成 Finger Print 的信息就会发生变化，这样我们就无法获取结果了。后文中我们还会再介绍该方法。

这里存在一个问题，那就是用户的网站访问记录相关的信息是个人信息。为了还原记录，每个用户都会被分配一个 ID。即使单独来看没有什么意义，但如果将信息组合起来能够推测出某个人，那么这些就是个人信息。Web 浏览器厂商正在加强个人隐私保护方面的功能，Safari 和 Firefox 都公布了针对追踪的策略。

Web 广告的历史是与安全相互磨合的历史。以能够完全明确到个人的方式获取用户信息的方法正在逐渐消失。退一步说，即使存在这样的方法，浏览器厂商也会填补该漏洞。广告分发正在成为一个需要专业人士来负责的领域。他们需要理解基于 Cookie 的统计工具和 Finger Print 方式等的特点，一边通过想象补充所获取的结果，一边进行网站的改进。

下面介绍一个移动应用程序的例子。过去在 Apple 的终端中存在终端固有的 ID—— UDID，许多企业用它来识别用户。现在，Apple 公司提供了用户可以自由重置的用于广告的广告标识符（Identification For Advertisers，广告标识符）。总体来说，Apple 公司采取了严格保护个人隐私的方针。

### 14.11.1 第三方 Cookie

第 2 章中介绍的 Cookie 称为第一方 Cookie。浏览器访问过的服务会写入仅在该服务内有效的 Cookie。由于写入源和读取源都受到限制，所以不会发生什么问题。不过，在广告等用途上，有的 Cookie 会填充从外部服务读取的 Cookie，以跟踪跨站点的行为。这就是第三方 Cookie。

第三方 Cookie 是与所访问的站点（A.com）不同的站点（B.com）的 Cookie，使用第三方 Cookie 的方法有在返回使用 `<img>` 标签引用的图像时包含 `Set-Cookie` 首部；根据 `<iframe>` 标签创建表单并发送；显示弹出式窗口；在 JavaScript 中使用 `postMessage()`，与打开其他站点的窗口交换信息.....

由于 B.com 的 Cookie 被保存了起来，所以当用户直接访问 B.com 时，Cookie 会被直接发送给 B.com 的服务器，这时就可以一并获取用户之前访问的信息。

还存在一种广告从业人员和用户无法直接访问 B.com 的情况。在这种情况下，可以使用 `<iframe>` 在 A.com 和 C.com 中识别用户。由于该 `<iframe>` 的内部与 B.com 一样，所以它们都可以发送 B.com 的 Cookie，这样就可以使用 Cookie 来识别跨页面的操作了。不过，A.com 和 C.com 的站点管理人员无法获得 B.com 的信息。

如果能够获得用户查看特定页面的信息，那么以此为线索，B.com 的服务器就会知道用户访问网站的倾向。与此相同，如果在各个站点的页面收集信息，统计同一个用户访问的页面记录，就能绘制更详细的用户画像。

在第三方 Cookie 中，像这样存在用户在没有意识到的情况下被追踪的个人隐私问题。Safari 中关于 Cookie 的默认设置是“允许访问过的网站的 Cookie”，拒绝除访问过的网站之外的所有域的 Cookie。2019 年 9 月，Firefox 也默认拦截第三方 Cookie。正如第 2 章中介绍的那样，Chrome 中也默认拦截未设置 `SameSite` 属性的第三方 Cookie。

### 14.11.2 Cookie 以外的代替手段

在识别客户端的方法中，最常用的就是在 Cookie 中记录会话令牌。对于 Cookie，我们可以根据安全强度选择忽略、重置或进行确认，不过也存在通过其他方法实现的难以重置的 Cookie，这种 Cookie 叫作“僵尸 Cookie”（Zombie Cookie）。还存在使用 JavaScript 来操作这些 Cookie 的 Evercookie 库。

HTML5 的各种存储（会话、本地、全局）

ETag

随机生成的图像

Flash、Silverlight、Java Applet 等存储区域

HTML5 的各种存储通过访问服务器以获取 ID，并将其保存到浏览器另行准备的存储中，来实现与 Cookie 相同的功能。在 ETag 的情况下，通过将用户 ID 放入肯定不会发生变化的图像文件等的 ETag 中并发送给客户端，来实现与 Cookie 相同的功能。客户端为了显示页面，会确认缓存是否有效，这时会向服务器发送 ETag 的信息，所以在不知不觉中就进行了与 Cookie 等价的处理。当通过 JavaScript 使用 Canvas 等时，由于可以从随机生成的图像中以像素为单位取出像素的颜色信息，所以可以将 ID 值作为多个像素的颜色信息进行存储。不过，按各个站点分别进行缓存的 Safari 不可以使用该方法。

这些结构可能会导致 ID 在不知不觉间被存储，进而使用户行为被监视，因此存在一些个人隐私方面的问题，比如通过安全设置来限制 Cookie 也没有用，或者无法将 Cookie 删除等。

### 14.11.3 GoogleAnalytics

除了广告之外，在其他一些情况下也需要识别用户，比如获取用户访问记录的 Google Analytics。在 Google Analytics 的情况下，网站中会设置 JavaScript。使用 JavaScript

代码，网站以在自己的域中运行为前提，将 ID 信息作为第一方 Cookie 保存到 Cookie 中。

**Google Analytics** 和 Web 广告的信息范围有很大不同。Google Analytics 只要（正确）获取所设置的域内部的用户行为就能达到目的。而 Web 广告则会超出所设置的服务器的范围，在各种服务之间传递用户 ID，来收集信息。

由于是第一方 Cookie，所以当访问 Google Analytics 时，用户 ID 不会被发送给 Google。由于能够从 HTML 的 `document.cookie` 中获取该信息，所以可以从 Google 提供的 JavaScript 代码进行读取，该 JavaScript 代码会作为模拟的第三方 Cookie 被发送给服务器。

#### 14.11.4 在不确定用户的情况下进行推测 (FingerPrint)

Experian 公司的 AdTruth 中使用的方法与通过 Cookie 等将 ID 存储到浏览器中来使用的方法不同。基于 IP 地址和浏览器的版本信息等能够从外部观测的信息来推测终端是否是同一个的方法，叫作 Finger Print 方式。与用户认证不同，Finger Print 方式只要大致得到用于广告的属性信息就可以了，不用识别具体用户，所以该方法很难应用于广告之外的情形。

客户端不持有任何信息，根据服务器推测的 ID 来分发广告，这种 Finger Print 方式在结构上接近于 GeoIP。根据 Experian 网站的内容，Finger Print 方式主要是以以下信息为基础的。

CPU、系统设置语言、用户代理、用户设置语言、默认语言集

浏览器应用程序代码名称、浏览器名称、版本、语言

浏览器插件

系统时间、屏幕设置、字体高度

域名、本地时间、时区

不过，该方法在美国能够很好地实施，而在日本实施时的精度就不是很高了。因为美国本土就有 4 个时区，使用的语言也有很多种。另外，Android 终端的使用比例相对较高，在根据属性信息分类后，总体就变小了。而日本只有 1 个时区，90% 以上的人使用日语，居住



地也都集中在大城市。另外，日本的智能手机占有率中 iPhone 的比例极高，因此推测起来比较困难。

浏览器名称和版本就是第 2 章介绍过的用户代理的 `User-Agent` 首部。在安全方面要求严格的 Apple 公司曾经试图将用户代理字符串固定，让人不知道详细的版本信息等，但后来取消了该做法<sup>9</sup>，其中一个原因就是 Finger Print。

## 14.12 本章小结

本章介绍了一些具有代表性的网络安全事件。与直接攻击计算机、夺取操作权限、盗取信息的以往的恶意软件不同，本章介绍的大部分攻击在计算机上完全不留痕迹。

本章仅介绍了通过 HTTP 首部等来应对攻击的事例。Web 安全问题由来已久，第 4 章介绍的跨站跟踪等问题过去发生过，但在现在的浏览器中并不会出现。在不牺牲便捷性的前提下，浏览器也在不断采取对策。正如本章介绍的那样，在安全与功能之间取得平衡，使用首部等来恰当地赋予权限，预防或者降低损失，浏览器正在朝着这个方向不断进化。

安全并不是服务器恰当地设置首部，浏览器准确进行实现就能保证的。如果恶意软件在计算机上设置代理服务器，同时登录根证书，就会引起中间人攻击。即使 Web 服务采取措施，也会存在很大的安全漏洞。

在编写本书的过程中，很多中间件和语言的库进行了紧急升级。我们需要适时对软件进行升级，不进行有可能造成安全漏洞的设置，准确地实现 Web 应用程序。有时也需要与提供安全诊断服务的公司合作，委托他们进行检查。

## AI智能总结

本文深入探讨了浏览器面临的各种安全攻击及相应的防范措施。重点介绍了会话令牌、Cookie的作用，以及针对跨站脚本攻击（XSS）的防范方法。此外，还详细讨论了针对资源文件和非资源文件设置的Content-Security-Policy指令，以及Mixed Content的应对策略和CORS的作用。另外，文章还提及了中间人攻击和会话劫持的危害以及相应的防范措施，包括使用HTTPS、HSTS等技术来抵御这些攻击。此外，还介绍了Cookie注入、跨站请求伪造、点击劫持和列表型账户入侵等攻击方式，并提出了相应的防范措施。文章还涉及了密码的保存、散列函

数的使用、密码的日志掩码化以及多因素身份验证等内容。总的来说，本文内容丰富，为读者提供了全面的浏览器安全防范策略和防范措施，对关注网络安全的技术人员具有很高的参考价值。

---

- [1]: 最近完全看不到该话题，但 Ruby 在很早之前就嵌入了基于该思想的污染标识、安全等级等功能。
- [2]: 引自德丸浩的博客文章《差的消毒、好的（？）消毒和异常处理》（原题为「悪いサニタイズ、良い(?) サニタイズ、そして例外処理」）。
- [3]: 引自 Chrome Platform Status 网站的“Strict Secure Cookies”一文。
- [4]: 这是不规范的行为，原本是没有这样的使用方法的。
- [5]: 注册流程麻烦是用户放弃使用服务的一个很大的原因，因此采用该方法能够增加用户量。
- [6]: 有时还包含位置因素。
- [7]: Kyash 是日本的一家数字银行创业公司。图中邮件的大意是“Kyash 被登录。如果你不知道此登录，请联系
- [8]: 最近的案例是发现了盗取 SSH 或 PGP 密钥的 Python 库。
- [9]: 引自「Safari の UA 文字列が固定されて固定されなくなったおはなし」（关于 Safari 的 UA 字符串固定和不再固定的话题）一文。

---

## 精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。