# Chapter 2
# Traditional Building Blocks of the Web

The previous chapter introduced the Web, covering the evolution from static Web pages towards a dynamic application platform. This chapter is more technical and provides the necessary background on how the Web works, which will help you in understanding the nuances of the attacks covered in the later chapters.

Within the distributed, hypertext-based Web, we will focus on the client-side features that enabled the Web to evolve into the dynamic application platform it is today and the browser security policies that are supposed to keep Web applications in line. Many of the topics covered in this chapter have been introduced in the early stages of the Web's development but are still present or reused in modern Web applications.

This chapter will first cover the basic building blocks of traditional Web applications, offering details on how content is loaded, how users can be authenticated and how session management mechanisms enhance the stateless HTTP protocol. Next, we will cover the browser's security policies, which regulate what Web applications can do within the browser, up to this day. We also investigate how client-side features can be extended beyond HTML, both by plugins for arbitrary content and by browser extensions. Finally, we cover several browser features that enhance the user's window on the Web.

## 2.1 Traditional Web Technology

Most modern Web applications are highly dynamic, process content in the background and fetch information on a continuous basis. While these applications seem vastly different from traditional Web applications, they share the same basis, and still use the same underlying concepts. This section briefly explains these traditional building blocks, offering you the required background knowledge.



P. De Ryck et al., *Primer on Client-Side Web Security,*
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-12226-7_2

### 2.1.1 Loading Web Content

Content on the Web is identified by a *Uniform Resource Identifier* (URI), a more general form of the earlier *Uniform Resource Locators* (URL). An example of a URI is *http://example.com/thisbook.html*. The first part of the URI, before the *://* is called the *scheme* and identifies the protocol to be used for fetching the resource. Most URIs on the Web today use the *http* or *https* scheme, which identifies HTTP protocol [18], either over a plaintext channel or over a secure channel, using Transport Layer Security [15], a topic that will be discussed in more detail in Chap. 5. Whenever the browser wants to load such a resource, it issues an HTTP request to the remote server, which is identified by the next part of the URI, between the *://* and the next */*, here *example.com*. The server at this address responds with an appropriate HTTP response for the requested path and parameters, which are the last part of the URI, here *thisbook.html*. This request/response-based communication protocol lies at the basis of every communication on the Web, even today.

HTTP requests and responses follow a certain pattern but have many configurable fields. A request has a certain method, such as GET, to retrieve information and POST to submit data to the server. In addition, both requests and responses can have headers, carrying meta-information about the request. We will not discuss all these possibilities in detail, information that can easily be found in other reference works [36]. One specific characteristic of HTTP that is relevant for the remainder of this text is that the protocol is *stateless*, meaning that there is no relation or required order between subsequent requests. Any need for relations or order between requests needs to be maintained by the browser and/or servers, on top of HTTP, for example by using request and response headers.

HTTP/2.0 [8], currently under development by the IETF, will bring several significant improvements to HTTP protocol, while preserving the original protocol's semantics. HTTP/2.0, based on the SPDY protocol by Google [7], essentially changes the way HTTP traffic is sent on the wire, reducing the page load time. HTTP/2.0 introduces new features such as multiplexed requests, prioritized requests, compressed headers, and support for server-pushed content. On the security side, a secure channel using TLS has not been made mandatory, but the most efficient upgrade path from the current HTTP/1.1 to HTTP/2.0 is by deploying HTTP/2.0 over TLS using the *Application Layer Protocol Negotiation* extension [19].

### 2.1.2 Authentication and Authorization

Even in the early Web, when sites offered only static content, authentication and authorization could be used to restrict access to the provided content. HTTP protocol provides the *Authorization* request header, aimed at providing the Web application with the user's credentials. The most common authentication scheme with the *Authorization* header uses *Basic* authentication, where the username and password are

base64-encoded,[1] and included as the header value. This allows an application to extract these credentials, verify them, and make a decision on whether to allow the request or not. After a successful authentication, the browser will attach the user's credentials to every subsequent request to this origin. Since the browser remembers these credentials during its session, logging out of an account is only possible by closing the browser.

As Web applications became more complex, developers wanted to integrate authentication with the application, streamlining the user experience within the same look and feel. To authenticate users, they embedded an HTML form, where the user had to enter a username and a password. By submitting the form, the username and password were sent to the server, where they could be validated. However, since the credentials are only sent in a single request after form submission, instead of in every subsequent request as with the *Authorization* header, Web applications needed a way to remember the user's authentication state. Keeping track of the authentication state is solved by using session management, our next topic of discussion.

### *2.1.3 Cookies and Session Management*

As mentioned before, subsequent requests in HTTP protocol are independent of each other. The lack of any relation between requests and the incapability of keeping state between requests has resulted in the introduction of cookies [4]. Cookies are server-provided key-value pairs, stored by the client, attached to every request to the same domain. Essentially, cookies allow the server to store some state at the client side, which will be attached to future requests. For example, cookies can be used to store a language preference, allowing the Web application to serve the requested content in the desired language.

A more complex mechanism, nowadays built on top of cookies, is session management. A session management mechanism offers a server-side session object, and associates multiple requests from the same user with this server-side session object. Web applications can use the session object to store useful session information, such as an authentication state, a shopping cart, etc.

Under the hood, session management mechanisms assign a random, unique session identifier to a newly created session. The session identifier is sent to the client in a cookie and will be attached to every subsequent request. By looking up the session object that belongs to the session identifier in the request, the Web application can process the request in the appropriate context.

Session management mechanisms based on session identifiers were already available before cookies were widely supported. These mechanisms included the session

[1] Base64 encoding transforms the entered username and password into an alphanumeric string, which is easily reversed. The credentials are not encrypted, as is often mistakenly believed.

identifier as a parameter in the URI, where it could be extracted by the Web application. These mechanisms have seen a slow demise because of practicality reasons, since every URI in the Web application needed to be dynamically generated to include the user-specific session identifier. In addition, embedding the identifier in the URI also holds a security risk, since the URI is easily leaked or copy/pasted. Note that many session management mechanisms still offer parameter-based session management as a fallback mechanism in case a browser does not support cookies.

## 2.2 Browser Security Policies

Modern browsers are the execution platform for complex Web applications, which consist of different kinds of static and dynamic content, coming from multiple providers with varying trust levels. Within the browser, several security policies govern the behavior of this content, regulating interactions between different contexts, managing access to potentially sensitive resources, and preventing unauthorized navigation attempts. These security policies are essential for client-side Web security, as their subtle nuances are often abused in attacks, and their restrictions are relied upon when building countermeasures.

Browser security policies generally depend on the notion of an *origin*. An origin is defined as the triple *(scheme, host, port)*, which are part of any URI,[2] in this section, we cover the three most important browser security policies. The *Same-Origin Policy* prevents unrestricted interactions between contexts from different origins, and regulates access to sensitive resources and application programming interfaces (APIs) on the basis of the origin of a document. This is the core security policy of the browser. Second, we discuss how the browser deals with the inclusion of cross-origin content, a common practice in almost every modern Web application. Third, we cover the *context navigation policy*, which is responsible for preventing unauthorized navigation requests between nested contexts.

### *2.2.1 Same-Origin Policy*

The core security policy in a browser is the Same-Origin Policy (SOP), which regulates direct interactions between different browsing contexts. The basic function of the SOP is to prevent scripts loaded in one origin from programmatically accessing resources from other origins. For example, if you have a script running on a page

[2] The port is an optional URI component, and when omitted, the protocol's default port is used, which is 80 for HTTP and 443 for HTTPS.

loaded from the URI *http://www.example.com*, it is not allowed to access the resources of a page loaded from *http://www.secret.com*, as the origins of both contexts are different, due to the distinct hosts.

There is one way to relax the constraints of the SOP, by using the *document.domain* JavaScript property, which allows two Web applications that share the same parent domain to interact with each other. For example, the application at *www.example.com* and *login.example.com* can both set their *document.domain* property to *example.com*, overriding any future same-origin checks with this parent domain. This allows two sibling applications to cooperate freely. Even though both parties must explicitly opt-in to this feature, once they have opted-in, any other site within the same parent domain can "join" as well.

The SOP originally started as a way to prevent access to the document object model (DOM) but has been gradually extended to other resources accessible within the origin. One such example is the recently introduced *canvas* element in HTML 5 [9], which allows the Web application to use JavaScript code to draw graphics and extract the result as an image. Certain features of the canvas allow the script to draw arbitrary, cross-origin resources on the canvas (e.g., an image or video), making it possible to steal the contents of the video or image. To prevent such cross-origin leaking, the specification requires the browser to consider the canvas to be "tainted" with the origin of the image, effectively preventing any access from an origin other than that of the image.

Another example is the XMLHttpRequest (XHR) object [35], which allows JavaScript code to issue new HTTP requests. Traditional XHR requests can only be sent to servers within the same origin as the origin of the document containing the script, motivated by the high degree of flexibility offered by the XHR object. Failing to restrict these requests would allow a malicious page to send custom HTTP requests to unsuspecting servers, for example, using the PUT or DELETE methods. As we will discuss in more detail in the next chapter, these restrictions have recently been relaxed with a server-driven security policy [34], inadvertently causing problems for applications that implicitly depended on this same-origin restriction [3].

Finally, the security policy for cookies is similar to origin-based policies but is actually domain-based. Cookies are generally set for a domain and only sent to the corresponding domain. A similar situation for script-based cookie access exists: any application that resides on the domain of the cookies, or a valid subdomain, can access the cookies from JavaScript. For example, cookies set explicitly for *www.example.com* can not only be accessed by any resource in *www.example.com* but also by resources under *dev.www.example.com*.

### *2.2.2 Security Model for Third-Party Content Inclusion*

Modern Web applications include content from a wide variety of locations, often residing within a different origin. Common examples of such third-party content inclusions are images, style sheets, or JavaScript files, simply integrated by including

an HTML tag with the appropriate URI. The inclusion of various JavaScript libraries is especially popular, since it allows the creation of highly responsive user interfaces, and enriches the Web site with additional functionality, ranging from integration with social media sites, to context-sensitive advertisements and tools for Web site analytics.

There are two commonly-used techniques to integrate third-party JavaScript into a Web application: through *script inclusion* or via *iframe integration*. The former loads the script within the security context of the including application, resulting in a straightforward way to integrate components and enable interaction between components. The latter places the script in a separate frame, within its own security context, effectively shielding sensitive resources, but making interaction a bit more complicated. We elaborate on both techniques below.

### Script Inclusion

HTML *script* tags are used to include and execute JavaScript while a Web page is loading. This JavaScript code can be located on a server with a different origin than the integrating page. When executing, the browser will treat the code as if it originated from the same origin as the Web page itself, without any restrictions of the SOP.

The included code executes in the same JavaScript context, and has access to the code of the integrating Web page and all of its data structures. All sensitive JavaScript operations available to the integrating Web page are also available to the integrated code.

**Prevalence of Third-Party Script Inclusion**

A 2012 study [25] examined 3,300,000 pages of the top 10,000 Alexa sites, and analyzed 8,439,799 remote script inclusions. From the results (shown in Fig. 2.1), it becomes clear that 88.45 % of the 10,000 Web sites included at least one remote JavaScript library. Even more remarkable, some sites in the top Alexa list trust up to 295 unique remote hosts.

The caveat that applies to third-party content inclusion is the interaction with the security policies of the browser, mainly the SOP. The included content generally resides directly within the security context of the including document, which is not a problem for static content, but results in complications when dynamic content, such as JavaScript, is included. The dynamic code is included within the security context of the application, where it gains access to all origin-restrained resources. Combined with the practice of including numerous third-party JavaScript libraries, this is a major security challenge for the Web.

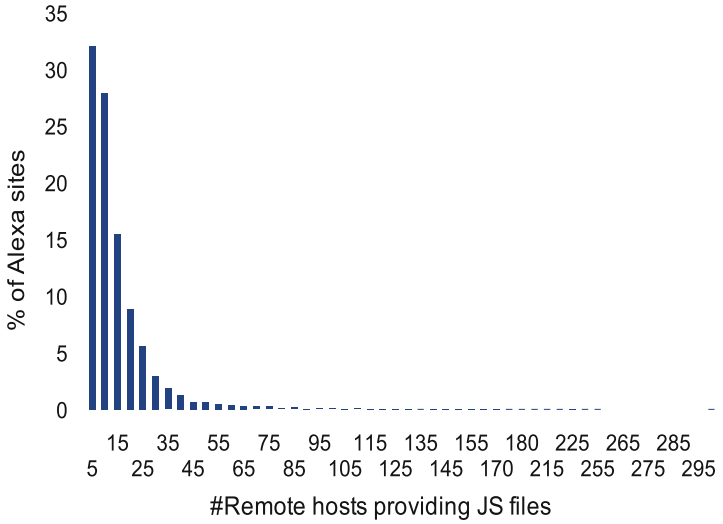


**Fig. 2.1** Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code. (Figure copyrighted by ACM, published in [25] with DOI *10.1145/2382196.2382274*)

#### Iframe Integration

HTML *iframe* tags allow a Web developer to include one document inside another. The integrated document is loaded in its own environment almost as if it were loaded in a separate browser window. The advantage of using an *iframe* in a Web application is that the integrated component (coming from another origin) is isolated from the integrating Web page by the SOP. However, the code running inside the *iframe* still has access to the available JavaScript APIs, albeit limited within its own execution context (i.e., origin). For instance, a third-party component can use local storage APIs but has access only to the local storage of its own origin and not to those of the integrating page.

The newly introduced HTML 5 *sandbox* attribute [10] aims to support the embedding of untrusted content in an *iframe* by putting security restrictions on the *iframe*, such as disabling JavaScript, turning off plugins, and restricting navigation. Through coarse-grained directives, several features can be re-enabled, for example by specifying the "allow-scripts" keyword to enable JavaScript.

### *2.2.3 Context Navigation Policy*

Navigation events occur frequently on the Web, for example when a user opens a page, follows a link, or when an automatic redirect happens. Triggering navigation events from within a document's context is straightforward, for example using

JavaScript to modify the *document.location* property or by automatically following a link. Navigation becomes more complicated when one context wants to navigate a window or frame from another context, possibly hosting a document from a different origin. Common examples are documents that want to navigate their child frames or a popup window they own. The decision as to whether to allow or deny such a navigation is not based on the SOP, which would prohibit any navigation between contexts from different origins, but is determined by a separate navigation policy. Several navigation policies have been proposed, but modern browsers all use the *descendant policy* [6, 9], which restricts navigation to child frames, or frames with an equivalent level of access [5].

## 2.3 Extending the Client-Side Features

Traditionally, browsers offer a rich set of features and ample functionality but also have limitations. By supporting extension mechanisms, browsers give developers the ability to enhance the browser experience, adding additional features. A first popular way of extending the browser is by adding plugins to handle arbitrary content. The most popular example of a browser plugin is Adobe's Flash player, which is capable of playing Flash files, which add dynamic content to a Web page. The second extension mechanism is browser extensions, which allow the user to modify the core behavior of the browser, adding additional features, or user interface (UI) items. Popular examples of browser extensions are NoScript, to limit the JavaScript that is run on a Web page, and AdBlock, to prevent intrusive advertisements from being loaded.

While these mechanisms clearly extend the functionality of the browser, they also have their consequences. For instance, they significantly enlarge the attack surface, as demonstrated by regular discoveries of malicious or vulnerable plugins or extensions [20, 31–33]. In this section, we briefly discuss how plugins and extensions work, how they are integrated in the browser and what consequences are associated with their use.

**Embedding Advertisements in a Web Application**

Many of the free Web applications are built on an advertisement-based business model, where third-party advertisements are embedded in the pages of the application. These advertisements are created by the companies that want to advertise their services or products, and are delivered through advertisement networks, such as DoubleClick, AdSense, and AdBrite.

Unfortunately, the Web application embedding the advertisements has no control of the content, and hence embeds untrusted content into its pages. Integrating untrusted content in a Web application comes with a trade-off between flexibility and security [13]. Using script inclusion for integrating the ads gives the advertisement provider great flexibility in ad placement, as well as

the capability to offer content-specific advertisements based on the displayed content of the embedding page. *iframe* integration offers the embedding page the necessary security guarantees but puts certain restrictions on the embedded content. The rigidity of *iframes* and the lack of security guarantees of scripts have driven the research community to come up with alternative approaches that offer isolation guarantees but enable the interaction demanded by advertisement networks [2, 16, 29]. From a high-level point of view, these approaches isolate the advertisement from the main page using a sandbox technique and allow a filtered set of interactions with the page's content. The most important interactions are drawing the advertisement in a dedicated part of the page and receiving the user's interaction with the advertisement, for example, clicking on the advertisement.

While these research efforts provide viable alternatives to the traditional *script* and *iframe*-based integration techniques, they are not used in practice. Almost every advertisement network uses *script*-based advertisements; this which occasionally results in the spreading of a malicious advertisement [21, 23, 27].

### 2.3.1 Plugins for Arbitrary Content

Browser plugins are generally designated handlers for specific kinds of content. The most common example of a browser plugin is Adobe's Flash Player, responsible for processing and displaying Flash content within the browser. Other popular examples are Silverlight, Java, ActiveX, and PDF reader plugins. Browser plugins are associated with MIME types and are automatically invoked when the content of a specified MIME type is encountered. The content is subsequently processed in the plugin's own runtime environment. A browser plugin can provide arbitrary functionality and is not limited to content rendering. An example of a non-content rendering plugin is the *Gnome Shell Integration* plugin [30], supporting the installation of additional widgets from the distribution site into the Gnome desktop environment.

Plugin content is embedded in a document, and the registered handler is triggered by the browser when this content is encountered. Most plugins allow communication between the document and the plugin using JavaScript, albeit with some restrictions, depending on the implementation. For example, in the case of Flash, an interface can be exposed towards the document, and arbitrary JavaScript functions can be executed in the embedding page.

Support for browser plugins is widespread on computing platforms running a traditional operating system, such as notebooks and desktop machines. Mobile support for browser plugins is extremely limited. For example, Apple's iOS does not support any browser plugins, and Adobe has abandoned efforts for supporting Flash on Android 4.1 and higher [11]. The demise of Flash on mobile platforms can mainly be

attributed to the rise of HTML 5's dynamic content features, and the performance issues associated with running a Flash player on a mobile device. Microsoft's Windows Phone only supports the in-house Silverlight plugin. Due to the limited support of plugins on mobile devices, Google has started to optimize search results, indicating which pages are likely to cause problems for your mobile device [37].

Plugin content runs within the environment of the handler, where the security policies of the browser no longer reign. This effectively means that if the plugin does not restrict the behavior of plugin content, basic browser policies are easily circumvented. One example is Flash, which allows developers to specify a policy to enable cross-origin requests, regardless of the SOP restrictions. A server can define the *crossdomain.xml* policy file, defining the origins from where remote requests are accepted. The Flash plugin is responsible for checking the file before carrying out the cross-origin request.

Plugins can also be a source of severe security problems, as illustrated by the numerous Java vulnerabilities in 2013 [31], eventually even leading to browser vendors recommending that Java should be disabled altogether. One potential source of vulnerabilities is the inability to deal with untrusted and potentially malicious input [36]. Therefore, close cooperation between browser vendors and plugin developers is crucial. In recent developments, the security of the Flash plugin has been significantly tightened. Flash is now effectively sandboxed on the OS level, preventing serious harm in case a vulnerability is found and exploited [22]. Alternatively, a new trend is emerging whereby plugin content is initially automatically disabled, but the user is then given the option to activate each piece of content separately, by a single click on a displayed *play* button [12].

### 2.3.2 Browser Extensions

Browser extensions extend the core functionality of the browser and come in various flavors, from a simple toolbar to behavior-changing extensions. An extension is not associated with a MIME content type but uses the exposed APIs to register hooks and react to events. Some popular examples of extensions are NoScript, which selectively disables JavaScript on Web pages; AdBlock, which removes advertisements; or FireBug, a Web development tool offering a debugger, giving a view on network traffic, etc.

Extensions for the Chrome and Firefox browsers are written in JavaScript,[3] and are restricted by the API offered by the browser. On Firefox, extensions can access almost all browser internals and can also access the file system or launch commands on the operating system. Chrome follows a more conservative approach, offering

[3] Native code is also supported but discouraged since it requires different versions for different platforms.

http://www.springer.com/978-3-319-12225-0

Primer on Client-Side Web Security

De Ryck, P.; Desmet, L.; Piessens, F.; Johns, M.

2014, X, 111 p. 13 illus., 12 illus. in color., Softcover
ISBN: 978-3-319-12225-0