Google

Code

Archive

browsersec - Part2.wiki

Export to GitHub

Browser Security Handbook, part 2

- Written and maintained by Michal Zalewski <lcamtuf@google.com>.
-
-

Table of Contents

Standard browser security features

This section provides a detailed discussion of explicit security mechanisms and restrictions implemented within browser. Long-standing design deficiencies are discussed, but no specific consideration is given to short-lived vulnerabilities.

Same-origin policy

Perhaps the most important security concept within modern browsers is the idea of the same-origin policy. The principal intent for this mechanism is to make it possible for largely unrestrained scripting and other interactions between pages served as a part of the same site (understood as having a particular DNS host name, or part thereof), whilst almost completely preventing any interference between unrelated sites.

In practice, there is no single same-origin policy, but rather, a set of mechanisms with some superficial resemblance, but quite a few important differences. These flavors are discussed below.

Same-origin policy for DOM access

With no additional qualifiers, the term "same-origin policy" most commonly refers to a mechanism that governs the ability for JavaScript and other scripting languages to access DOM properties and methods across domains (reference). In essence, the model boils down to this three-step decision process:

- If protocol, host name, and - for browsers other than Microsoft Internet Explorer - port number for two interacting pages match, access is granted with no further checks.
- Any page may set `document.domain` parameter to a right-hand, fully-qualified fragment of its current host name (e.g., `foo.bar.example.com` may set it to `example.com`, but not `ample.com`). If two pages explicitly and *mutually* set their respective `document.domain` parameters to the same value, and the remaining same-origin checks are satisfied, access is granted.
- If neither of the above conditions is satisfied, access is denied.

In theory, the model seems simple and robust enough to ensure proper separation between unrelated pages, and serve as a method for sandboxing potentially untrusted or risky content within a particular domain; upon closer inspection, quite a few drawbacks arise, however:

- Firstly, the `document.domain` mechanism functions as a security tarpit: once any two legitimate subdomains in `example.com`, e.g. `www.example.com` and `payments.example.com`, choose to cooperate this way, any other resource in that domain, such as `user-pages.example.com`, may then set own `document.domain` likewise, and arbitrarily mess with `payments.example.com`. This means that in many scenarios, `document.domain` may not be used safely at all.

- Whenever `document.domain` cannot be used - either because pages live in completely different domains, or because of the aforementioned security problem - legitimate client-side communication between, for example, embeddable page gadgets, is completely forbidden in theory, and in practice very difficult to arrange, requiring developers to resort to the abuse of known browser bugs, or to latency-expensive server-side channels, in order to build legitimate web applications.

- Whenever tight integration of services within a single host name is pursued to overcome these communication problems, because of the inflexibility of same-origin checks, there is no usable method to sandbox any untrusted or particularly vulnerable content to minimize the impact of security problems.

On top of this, the specification is simplistic enough to actually omit quite a few corner cases; among other things:

- The `document.domain` behavior when hosts are addressed by IP addresses, as opposed to fully-qualified domain names, is not specified.

- The `document.domain` behavior with extremely vague specifications (e.g., `com` or `co.uk`) is not specified.

- The algorithms of context inheritance for pseudo-protocol windows, such as `about:blank`, are not specified.

- The behavior for URLs that do not meaningfully have a host name associated with them (e.g., `file://`) is not defined, causing some browsers to permit locally saved files to access every document on the disk or on the web; users are generally not aware of this risk, potentially exposing themselves.

- The behavior when a single name resolves to vastly different IP addresses (for example, one on an internal network, and another on the Internet) is not specified, permitting DNS rebinding attacks and related tricks that put certain mechanisms (captchas, ad click tracking, etc) at extra risk.

- Many one-off exceptions to the model were historically made to permit certain types of desirable interaction, such as the ability to point own frames or script-spawned windows to new locations - and these are not well-documented.

All this ambiguity leads to a significant degree of variation between browsers, and historically, resulted in a large number of browser security flaws. A detailed analysis of DOM actions permitted across domains, as well as context inheritance rules, is given in later sections. A quick survey of several core same-origin differences between browsers is given below:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | May `document.domain` be set to TLD alone? | NO | NO | NO | YES | NO | YES | NO | YES | YES | | May `document.domain` be set to TLD with a trailing dot? | YES | YES | NO | YES | NO | YES | NO | YES | YES | | May `document.domain` be set to right-hand IP address fragments? | YES | YES | NO | YES | NO | YES | NO | NO | YES | | Do port numbers wrap around in same origin checks? | NO | NO | NO | uint32 | uint32 | uint16/32 | uint16 | NO | n/a | | May local HTML access unrelated local files via DOM? | YES | YES | YES | YES | NO | NO | YES | NO | n/a | | May local HTML access sites on the Internet via DOM? | NO | NO | NO | NO | NO | NO | NO | NO | n/a |

Note: Firefox 3 is currently the only browser that uses a directory-based scoping scheme for same-origin access within `file://`. This bears some risk of breaking quirky local applications, and may not offer protection for shared download directories, but is a sensible approach otherwise.

Same-origin policy for XMLHttpRequest

On top of scripted DOM access, all of the contemporary browsers also provide the XMLHttpRequest JavaScript API, by which scripts may make HTTP requests to their originating site, and read back data as needed. The mechanism was originally envisioned primarily to make it possible to read back XML responses (hence the name, and the `responseXML` property), but currently, is perhaps more often used to read back JSON messages, HTML, and arbitrary custom communication protocols, and serves as the foundation for much of the *web 2.0* behavior of rapid UI updates not dependent on full-page transitions.

The set of security-relevant features provided by `XMLHttpRequest`, and not seen in other browser mechanisms, is as follows:

- The ability to specify an arbitrary HTTP request method (via the `open()` method),
- The ability to set custom HTTP headers on a request (via `setRequestHeader()`),
- The ability to read back full response headers (via `getResponseHeader()` and `getAllResponseHeaders()`),
- The ability to read back full response body as JavaScript string (via `responseText` property).

Since all requests sent via `XMLHttpRequest` include a browser-maintained set of cookies for the target site, and given that the mechanism provides a far greater ability to interact with server-side components than any other feature available to scripts, it is extremely important to build in proper security controls. The set of checks implemented in all browsers for `XMLHttpRequest` is a close variation of DOM same-origin policy, with the following changes:

- Checks for `XMLHttpRequest` targets do not take `document.domain` into account, making it impossible for third-party sites to mutually agree to permit cross-domain requests between them.

- In some implementations, there are additional restrictions on protocols, header fields, and HTTP methods for which the functionality is available, or HTTP response codes which would be shown to scripts (see later).
- In Microsoft Internet Explorer, although port number is not taken into account for "proper" DOM access same-origin checks, it is taken into account for `XMLHttpRequest`.

Since the exclusion of `document.domain` made any sort of client-side cross-domain communications through `XMLHttpRequest` impossible, as a much-demanded extension, W3C proposal for cross-domain XMLHttpRequest access control would permit cross-site traffic to happen under certain additional conditions. The scheme envisioned by the proponents is as follows:

- `GET` requests with custom headers limited to a whitelist would be sent to the target system immediately, with no advance verification, based on the assumption that `GET` traffic is not meant to change server-side application state, and thus will have no lasting side effects. This assumption is theoretically sound, as per the "SHOULD NOT" recommendation spelled out in RFC 2616, though is seldom observed in practice. Unless an appropriate HTTP header or XML directive appears in the response, the result would not be revealed to the requester, though.
- Non-`GET` requests (`POST`, etc) would be preceded by a "preflight" `OPTIONS` request, again with only whitelisted headers permitted. Unless an appropriate HTTP header or XML directive is seen in response, the actual request would not be issued.

Even in its current shape, the mechanism would open some RFC-ignorant web sites to new attacks; some of the earlier drafts had more severe problems, too. As such, the functionality ended up being scrapped in Firefox 3, and currently, is not available in any browser, pending further work. A competing proposal from Microsoft, making an Microsoft Internet Explorer 8, implements a completely incompatible, safer, but less useful scheme - permitting sites to issue anonymous (cookie-less) cross-domain requests only. There seems to be an ongoing feud between these two factions, so it may take a longer while for any particular API to succeed, and it is not clear what security properties it would posses.

As noted earlier, although there is a great deal of flexibility in what data may be submitted via `XMLHttpRequest` to same-origin targets, various browsers blacklist subsets of HTTP headers to prevent ambiguous or misleading requests from being issued to servers and cached by the browser or by any intermediaries . These restrictions are generally highly browser-specific; for some common headers, they are as follows:

| **HTTP header** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | `Accept` | OK | OK | OK | OK | OK | OK | OK | OK | OK | | `Accept-Charset` | OK | OK | OK | OK | BANNED | BANNED | BANNED | BANNED | BANNED | | `Accept-Encoding` | BANNED | BANNED | BANNED | OK | BANNED | BANNED | BANNED | BANNED | BANNED | | `Accept-Language` | OK | OK | OK | OK | OK | OK | OK | BANNED | BANNED | | `Cache-Control` | OK | OK | OK | OK | OK | OK | BANNED | OK | OK | | `Cookie` | BANNED | BANNED | BANNED | OK | OK | BANNED | BANNED | BANNED | OK | | `If-*` family (`If-Modified-Since`, etc) | OK | OK | OK | OK | OK | OK | BANNED | OK | OK | | `Host` | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | | `Range` | OK | OK | OK | OK | OK | OK | BANNED | OK | OK | | `Referer` | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | | `Transfer-Encoding` | OK | OK | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED | | `User-Agent` | OK | OK | OK | OK | OK | BANNED | OK | BANNED | BANNED | | `Via` | OK | OK | OK | BANNED | BANNED | BANNED | BANNED | BANNED | BANNED |

Specific implementations may be examined for a complete list: the current WebKit trunk implementation can be found here, whereas for Firefox, the code is here.

A long-standing security flaw in Microsoft Internet Explorer 6 permits stray newline characters to appear in some `XMLHttpRequest` fields, permitting arbitrary headers (such as `Host`) to be injected into outgoing requests. This behavior needs to be accounted for in any scenarios where a considerable population of legacy MSIE6 users is expected.

Other important security properties of `XMLHttpRequest` are outlined below:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Banned HTTP methods | `TRACE` | `CONNECT TRACE` | *`CONNECT TRACE`* | `TRACE` | `TRACE` | `CONNECT TRACE` | `CONNECT TRACE`** | `CONNECT TRACE` | `CONNECT TRACE` | | `XMLHttpRequest` may see `httponly` cookies? | NO | NO | NO | YES | NO | YES | NO | NO | NO | | `XMLHttpRequest` may see invalid HTTP 30x responses? | NO | NO | NO | YES | YES | NO | NO | YES | NO | | `XMLHttpRequest` may see cross-domain HTTP 30x responses? | NO | NO | NO | YES | YES | NO | NO | NO | NO | | `XMLHttpRequest` may see other HTTP non-200 responses? | YES | YES | YES | YES | YES | YES | YES | YES | NO | | May local HTML access unrelated local files via `XMLHttpRequest`? | NO | NO | NO | YES | NO | NO | YES | NO | n/a | | May local HTML access sites on the Internet via `XMLHttpRequest`? | YES | YES | YES | NO | NO | NO | NO | NO | n/a | | Is partial `XMLHttpRequest` data visible while loading? | NO | NO | NO | YES | YES | YES | NO | YES | NO |

Implements a whitelist of known schemes, rejects made up values.
* Implements a whitelist of known schemes, replaces non-whitelisted schemes with `GET`.

WARNING: Microsoft Internet Explorer 7 may be forced to partly regress to the less secure behavior of the previous version by invoking a proprietary, legacy `ActiveXObject('MSXML2.XMLHTTP')` in place of the new, native `XMLHttpRequest` API.

Please note that the degree of flexibility offered by `XMLHttpRequest`, and not seen in other cross-domain content referencing schemes, may be actually used as a simple security mechanism: a check for a custom HTTP header may be carried out on server side to confirm that a cookie-authenticated request comes from JavaScript code that invoked `XMLHttpRequest.setRequestHeader()`, and hence must be triggered by same-origin content, as opposed to a random third-party site. This provides a coarse cross-site request forgery defense, although the mechanism may be potentially subverted by the incompatible same-origin logic within some plugin-based programming languages, as discussed later on.

Same-origin policy for cookies

As the web started to move from static content to complex applications, one of the most significant problems with HTTP was that the protocol contained no specific provisions for maintaining any client-associated context for subsequent requests, making it difficult to implement contemporary mechanisms such as convenient, persistent authentication or preference management (HTTP authentication, as discussed later on, proved to be too cumbersome for this purpose, while any in-URL state information would be often accidentally disclosed to strangers or lost). To address the need, HTTP cookies were implemented in Netscape Navigator (and later captured in spirit as RFC 2109, with neither of the standards truly followed by most implementations): any server could return a short text token to be stored by the client in a `Set-Cookie` header, and the token would be stored by clients and included on all future requests (in a `Cookie` header).

Key properties of the mechanism:

- **Header structure:** in theory, every `Set-Cookie` header sent by the server consists of one or more comma-separated `NAME=VALUE` pairs, followed by a number of additional semicolon-separated parameters or keywords. In practice, a vast majority of browsers support only a single pair (confusingly, multiple `NAME=VALUE` pairs may accepted in some browsers via `document.cookie`, a simple JavaScript cookie manipulation API). Every `Cookie` header sent by the client consists of any number of semicolon-separated `NAME=VALUE` pairs with no additional metadata.

- **Scope:** by default, cookie scope is limited to all URLs on the current host name - and **not** bound to port or protocol information. Scope may be limited with `path=` parameter to specify a specific path prefix to which the cookie should be sent, or broadened to a group of DNS names, rather than single host only, with `domain=`. The latter operation may specify any fully-qualified right-hand segment of the current host name, up to one level below TLD (in other words, `www.foo.bar.example.com` may set a cookie to be sent to `*.bar.example.com` or `*.example.com`, but not to `*.something.else.example.com` or `*.com`); the former can be set with no specific security checks, and uses just a dumb left-hand substring match.*Note: according to one of the specs, domain wildcards should be marked with a preceeding period, so `.example.com` would denote a wildcard match for the entire domain - including, somewhat confusingly, `example.com` proper - whereas `foo.example.com` would denote an exact host match. Sadly, no browser follows this logic, and `domain=example.com` is exactly equivalent to `domain=.example.com`. There is no way to limit cookies to a single DNS name only, other than by not specifying `domain=` value at all - and even this does not work in Microsoft Internet Explorer; likewise, there is no way to limit them to a specific port.*

- **Time to live:** by default, each cookie has a lifetime limited to the duration of the current browser session (in practice meaning that it is stored in program memory only, and not written to disk). Alternatively, an `expires=` parameter may be included to specify the date (in one of a large number of possible confusing and hard-to-parse date formats) at which the cookie should be dropped. This automatically enables persistent storage of the cookie. A much less commonly used, but RFC-mandated `max-age=` parameter might be used to specify expiration time delta instead.

- **Overwriting cookies:** if a new cookie with the same `NAME`, `domain`, and `path` as an existing cookie is encountered, the old cookie is discarded. Otherwise, even if a subtle difference exists (e.g., two distinct `domain=` values in the same top-level domain), the two cookies will co-exist, and may be sent by the client at the same time as two separate pairs in `Cookie` headers, with no additional information to help resolve the conflict.

- **Deleting cookies:** There is no specific mechanism for deleting cookies envisioned, although a common hack is to overwrite a cookie with a bogus value as outlined above, plus a backdated or short-lived `expires=` (using `max-age=0` is not universally supported).

- **"Protected" cookies:** as a security feature, some cookies set may be marked with a special `secure` keyword, which causes them to be sent over HTTPS only. Note that non-HTTPS sites may still set `secure` cookies in some implementations, just not read them back.

The original design for HTTP cookies has multiple problems and drawbacks that resulted in various security problems and kludges to address them:

- **Privacy issues:** the chief concern with the mechanism was that it permitted scores of users to be tracked extensively across any number of collaborating domains without permission (in the simplest form, by simply including tracking code in an `IFRAME` pointing to a common `evil-tracking.com` resource on any number of web pages, so that the same `evil-tracking.com` cookie can be correlated across all properties). It is a major misconception that HTTP cookies were the only mechanism to store and retrieve long-lived client-side tokens - for example, cache validation directives or `window.name` DOM property may be naughtily repurposed to implement a very similar functionality - but the development nevertheless caused public outcry.Widespread criticism eventually resulted in many browsers enabling restrictions on any included content on a page setting cookies for any domain other than that displayed in the URL bar (discussed later on), despite the fact that such a measure would not stop cooperating sites from tracking users using marginally more sophisticated methods. A minority of users to this day browses with cookies disabled altogether for similar reasons, too.

- **Problems with ccTLDs:** the specification did not account for the fact that many country-code TLDs are governed by odd or sometimes conflicting rules. For example, `waw.pl`, `com.pl`, and `co.uk` should be all seen as generic, functional top-level domains, and so it should not be possible to set cookies at this level, as to avoid interference between various applications; but `example.pl` or `coredump.cx` are single-owner domains for which it

should be possible to set cookies. This resulted in many browsers having serious trouble collecting empirical data from various ccTLDs and keeping it in sync with the current state of affairs in the DNS world.

- **Problems with conflict resolution:** when two identically named cookies with different scopes are to be sent in a single request, there is no information available to the server to resolve the conflict and decide which cookie came from where, or how old it is. Browsers do not follow any specific conventions on the ordering of supplied cookies, too, and some behave in an outright buggy manner. Additional metadata to address this problem is proposed in "cookies 2" design (RFC 2965), but the standard never gained widespread support.

- **Problems with certain characters:** just like HTTP, cookies have no specific provisions for character escaping, and no specified behavior for handling of high-bit and control characters. This sometimes results in completely unpredictable and dangerous situations if not accounted for.

- **Problems with cookie jar size:** standards do relatively little to specify cookie count limits or pruning strategies. Various browsers may implement various total and per-domain caps, and the behavior may result in malicious content purposefully disrupting session management, or legitimate content doing so by accident.

- **Perceived Java**Script-related problems:**the aforementioned `document.cookie` Java**Script API permits for JavaScript embedded on pages to access sensitive authentication cookies. If malicious scripts may be planted on a page due to insufficient escaping of user input, these cookies could be stolen and disclosed to the attacker. The concern for this possibility resulted in httponly cookie flag being incorporated into Microsoft Internet Explorer, and later other browsers; such cookies would not be visible through `document.cookie` (but, as noted in the previous section, are not always adequately hidden in `XMLHttpRequest` calls). In reality, the degree of protection afforded this way is minimal, given the ability to interact with same-origin content through DOM.

- **Problems with "protected" cookie clobbering:** as indicated earlier, `secure` and `httponly` cookies are meant not to be visible in certain situations, but no specific thought was given to preventing JavaScript from overwriting `httponly` cookies, or non-encrypted pages from overwriting `secure` cookies; likewise, `httponly` or `secure` cookies may get dropped and replaced with evil versions by simply overflowing the per-domain cookie jar. This oversight could be abused to subvert at least some usage scenarios.

- **Conflicts with DOM same-origin policy rules:** cookies have scoping mechanisms that are broader and essentially incompatible with same-origin policy rules (e.g., as noted, no ability to restrict cookies to a specific host or protocol) - sometimes undoing some content security compartmentalization mechanisms that would otherwise be possible under DOM rules.

An IETF effort is currently underway to clearly specify currently deployed cookie behavior across major browsers.

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Does `document.cookie` work on `ftp` URLs? | NO | NO | NO | NO | NO | NO | NO | n/a | | Does `document.cookie` work on `file` URLs? | YES | YES | YES | YES | YES | YES | YES | NO | n/a | | Is `Cookie2` standard supported? | NO | NO | NO | NO | NO | NO | YES | NO | NO | | Are multiple comma-separated `Set-Cookie` pairs accepted? | NO | NO | NO | NO | NO | YES | NO | NO | NO | | Are `quoted-string` values supported for HTTP cookies? | NO | NO | NO | YES | YES | NO | YES | NO | YES | | Is `max-age` parameter supported? | NO | NO | NO | YES | YES | YES | YES | YES | YES | | Does `max-age=0` work to delete cookies? | (NO) | (NO) | (NO) | YES | YES | NO | YES | YES | YES | | Is `httponly` flag supported? | YES | YES | YES | YES | YES | YES | YES | YES | NO | | Can scripts clobber `httponly` cookies? | *NO* | *NO* | *NO* | *YES* | *NO* | *YES* | *NO* | *NO* | *(YES)* | | *Can HTTP pages clobber `secure` cookies?* | YES | YES | YES | YES | YES | YES | YES | YES | YES | | Ordering of duplicate cookies with different scope | random | random | some dropped | some dropped | most specific first | random | most specific first | most specific first | by age | | Maximum length of a single cookie | 4 kB | 4 kB | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | broken | | Maximum number of cookies per site | 50 | 50 | 50 | ∞ | 100 | ∞ | ∞ | 150 | 50 | | Are cookies for right-hand IP address fragments accepted? | NO | NO | NO | NO | NO | YES | NO | NO | NO | | Are host-scope cookies possible (no `domain=` value)? | NO | NO | NO | YES | YES | YES | YES | YES | YES | | Overly permissive ccTLD behavior test results (3 tests) | 1/3 FAIL | 1/3 FAIL | 3/3 OK | 2/3 FAIL | 3/3 OK | 1/3 FAIL | 3/3 OK | 3/3 OK | 2/3 FAIL |

* Note that as discussed earlier, even when this is not directly permitted, the attacker may still drop the original cookie by simply overflowing the cookie jar, and insert a new one without a `httponly` or `secure` flag set; and even if the ability to overflow the jar is limited, there is no way for a server to distinguish between a genuine `httponly` or `secure` cookie, and a differently scoped, but identically named lookalike.

Same-origin policy for Flash

Adobe Flash, a plugin believed to be installed on about 99% of all desktops, incorporates a security model generally inspired by browser same-origin checks. Flash applets have their security context derived from the URL they are loaded from (as opposed to the site that embeds them with `<OBJECT>` or `<EMBED>` tags), and within this realm, permission control follows the same basic principle as applied by browsers to DOM access: protocol, host name, and port of the requested resource is compared with that of the requestor, with universal access privileges granted to content stored on local disk. That said, there are important differences - and some interesting extensions - that make Flash capable of initiating cross-domain interactions to a degree greater than typically permitted for native browser content.

Some of the unique properties and gotchas of the current Flash security model include:

- The ability for sites to provide a cross-domain policy, often referred to as `crossdomain.xml`, to allow a degree of interaction from non-same-origin content. Any non-same-origin Flash applet may specify a location on the target server at which this XML-based specification should be looked up; if it

matches a specific format, it would be interpreted as a permission to carry out cross-domain actions for a given target URL path and its descendants.Historically, the mechanism, due to extremely lax XML parser and no other security checks in place, posed a major threat: many types of user content, for example images or text files, could be trivially made to mimick such data without site owner's knowledge or consent. Recent security improvements enabled a better control of cross-domain policies; this includes a more rigorous XML parser; a requirement for MIME type on policies to match `text/*`, `application/xml`, or `application/xhtml+xml`; or the concept of site-wide meta-policies, stored at a fixed top-level location - `/crossdomain.xml`. These policies would specify global security rules, and for example prevent any lower-order policies from being interpreted, or require MIME type on all policies to non-ambiguously match `text/x-cross-domain-policy`.

- The ability to make cookie-bearing cross-domain HTTP `GET` and `POST` requests via the browser stack, with fewer constraints than typically seen elsewhere in browsers. This is achieved through the URLRequest API. The functionality, most notably, includes the ability to specify arbitrary `Content-Type` values, and to send binary payloads. Historically, Flash would also permit nearly arbitrary headers to be appended to cross-domain traffic via the `requestHeaders` property, although this had changed with a series of recent security updates, now requiring an explicit `crossdomain.xml` directive to re-enable the feature.

- The ability to make same-origin HTTP requests, including setting and reading back HTTP headers to an extent greater than that of `XMLHttpRequest` (list of banned headers).

- The ability to access to raw TCP sockets via `XMLSockets`, to connect back to the same-origin host on any high port (> 1024), or to access third-party systems likewise. Following recent security updates, this requires explicit cross-domain rules, although these may be easily provided for same-origin traffic. In conjunction with DNS rebinding attacks or the behavior of certain firewall helpers, the mechanism could be abused to punch holes in the firewall or probe local and remote systems, although certain mitigations were incorporated since then.

- The ability for applet-embedding pages to restrict certain permissions for the included content by specifying `<OBJECT>` or `<EMBED>` parameters:

  - The ability to load external files and navigate the current browser window (allowNetworking attribute).
  - The ability to interact with on-page JavaScript context (allowScriptAccess attribute; previously unrestricted by default, now limited to `sameDomain`, which requires the accessed page to be same origin with the applet).
  - The ability to run in full-screen mode (allowFullScreen attribute).

This model is further mired with other bugs and oddities, such as the reliance on location.* DOM being tamper-proof for the purpose of executing same-origin security checks.

Flash applets running from the Internet do not have any specific permissions to access local files or input devices, although depending on user configuration decisions, some or all sites may use a limited quota within a virtualized data storage sandbox, or access the microphone.

Same-origin policy for Java

Much like Adobe Flash, Java applets, reportedly supported on about 80% of all desktop systems, roughly follow the basic concept of same-origin checks applied to a runtime context derived from the site the applet is downloaded from - except that rather unfortunately to many classes of modern websites, different host names sharing a single IP address are considered same-origin under certain circumstances.

The documentation for Java security model available on the Internet appears to be remarkably poor and spotty, so the information provided in this section is in large part based on empirical testing. According to this research, the following permissions are available to Java applets:

- The ability to interact with JavaScript on the embedding page through the JSObject API, with no specific same-origin checks. This mechanism is disabled by default, but may be enabled with the `MAYSCRIPT` parameter within the `<APPLET>` tag.

- In some browsers, the ability to interact with the embedding page through the DOMService API. The documentation does not state what, if any, same-origin checks should apply; based on the aforementioned tests, no checks are carried out, and cross-domain embedding pages may be accessed freely with no need for `MAYSCRIPT` opt-in. This directly contradicts the logic of `JSObject` API.

- The ability to send same-origin HTTP requests using the browser stack via the URLConnection API, with virtually no security controls, including the ability to set `Host` headers, or insert conflicting caching directives. On the upside, it appears that there is no ability to read 30x redirect bodies or `httponly` cookies from within applets.

- The ability to initiate unconstrained TCP connections back to the originating host, and that host only, using the Socket API. These connections do not go through the browser, and are not subject to any additional security checks (e.g., ports such as `25/tcp` are permitted).

Depending on the configuration, the user may be prompted to give signed applets greater privileges, including the ability to read and write local files, or access specific devices. Unlike Flash, Java has no cross-domain policy negotiation features.

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| Is `DOMService` supported? | YES | YES | YES | NO | NO | YES | NO | YES | n/a | | Does `DOMService` permit cross-domain access to embedding page? | YES | YES | YES | n/a | n/a | YES | n/a | YES | n/a |

Same-origin policy for Silverlight

Microsoft Silverlight 2.0 is a recently introduced content rendering browser plugin, and a competitor to Adobe Flash.

There is some uncertainty about how likely the technology is to win widespread support, and relatively little external security research and documentation available at this time, so this section will be likely revised and extended at a later date. In principle, however, Silverlight appears to closely mimick the same-origin model implemented for Flash:

- Security context for the application is derived from the URL the applet is included from. Access to the embedding HTML document is permitted by default for same-origin HTML, and controlled by enableHtmlAccess parameter elsewhere. Microsoft security documentation does not clearly state if scripts have permission to navigate browser windows in absence of `enableHtmlAccess`, however.
- Same-origin HTTP requests may be issued via `HttpWebRequest` API, and may contain arbitrary payloads - but there are certain restrictions on which HTTP headers may be modified. The exact list of restricted headers is available here.
- Cross-domain HTTP and network access is not permitted until a policy compatible with Flash `crossdomain.xml` or Silverlight `clientaccesspolicy.xml` format, is furnished by the target host. Microsoft documentation implies that "unexpected" MIME types are rejected, but this is not elaborated upon; it is also not clear how strict the parser used for XML policies is (reference).
- Non-HTTP network connectivity uses `System.Net.Sockets`. Raw connections to same-origin systems are not permitted until an appropriate cross-domain policy is furnished (reference).
- Cross-scheme access between HTTP and HTTPS is apparently considered same-origin, and does not require a cross-domain specification (reference).

Same-origin policy for Gears

Google Gears is a browser extension that enables user-authorized sites to store persistent data in a local database. Containers in the database are partitioned in accordance with the traditional understanding of same-origin rules: that is, protocol, host name, and port must match precisely for a page to be able to access a particular container - and direct fenceposts across these boundaries are generally not permitted (reference).

An important additional feature of Gears are JavaScript workers: a specialized WorkerPool API permits authorized sites to initiate background execution of JavaScript code in an inherited security context without blocking browser UI. This functionality is provided in hopes of making it easier to develop rich and CPU-intensive offline applications.

A somewhat unusual, indirect approach to cross-domain data access in Gears is built around the `createWorkerFromUrl` function, rather than any sort of cross-domain policies. This API call permits a previously authorized page in one domain to spawn a worker running in the context of another; both the security context, and the source from which the worker code is retrieved, is derived from the supplied URL. For security reasons, the data must be further served with a MIME type of `application/x-gears-worker`, thus acknowledging mutual consent to this interaction.

Workers behave like separate processes: they do not share any execution state with each other or with their parent - although they may communicate with the "foreground" JavaScript code and workers in other domain through a simple, specialized messaging mechanism.

Workers also do not have access to a native `XMLHttpRequest` implementation, so Gears provides a compatible subset of this functionality through own HttpRequest interface. `HttpRequest` implementation blacklists a standard set of HTTP headers and methods, as listed in httprequest.cc.

Origin inheritance rules

As hinted earlier, certain types of pseudo-URLs, such as `javascript:`, `data:`, or `about:blank`, do not have any inherent same-origin context associated with them the way `http://` URLs have - which poses a special problem in the context of same-origin checks.

If a shared "null" security context is bestowed upon all these resources, and checks against this context always succeed, a risk arises that blank windows spawned by completely unrelated sites and used for different purposes could interfere with each other. The possibility is generally prevented in most browsers these days, but had caused a fair number of problems in the past (see Mozilla bug 343168 and related work by Adam Barth and Collin Jackson for a historical perspective). On the other hand, if all access to such windows is flat out denied, this would go against the expectation of legitimate sites to be able to scriptually access own `data:` or `about:blank` windows.

Various browsers accommodate this ambiguity in different ways, often with different rules for document-embedded `<IFRAME>` containers, for newly opened windows, and for descendants of these windows.

A quick survey of implementations is shown below:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Inherited context for empty IFRAMEs | parent | parent | parent | parent | parent | parent | parent | parent | | | Inherited context for `about:blank` windows | parent | parent | parent

| no access | no access | parent | parent | parent | parent | | Inherited context for `javascript:` windows | parent | parent | parent | parent | parent | n/a | parent | n/a | n/a | | Inherited context for `data:` windows | n/a | n/a | n/a | parent | parent | no access | blank | no access | no access | | Is parent's `Referer` sent from empty IFRAMEs? | NO | NO | NO | NO | NO | NO | YES | NO | NO | | Is parent's `Referer` sent from `javascript:` windows? | NO | NO | NO | NO | NO | n/a | NO | n/a | n/a | | Is parent's `Referer` sent from `data:` windows? | n/a | n/a | n/a | NO | NO | NO | NO | NO | NO |

Cross-site scripting and same-origin policies

Same-origin policies are generally perceived as one of most significant bottlenecks associated with contemporary web browsers. To application developers, the policies are too strict and inflexible, serving as a major annoyance and stifling innovation; the developers push for solutions such as cross-domain `XMLHttpRequest` or `crossdomain.xml` in order to be able to build and seamlessly integrate modern applications that span multiple domains and data feeds. To security engineers, on the other hand, these very same policies are too loose, putting user data at undue risk in case of minor and in practice nearly unavoidable programming errors.

The security concern traces back to the fact that the structure of HTML as such, and the practice for rendering engines to implement very lax and poorly documented parsing of legacy HTML features, including extensive and incompatible error recovery attempts, makes it difficult for web site developers to render user-controlled information without falling prey to HTML and script injection flaws. A number of poorly designed, browser-side content sniffing and character set strategies (discussed later in this document), further contributes to the problem by making it likely for non-HTML content to be misinterpreted and rendered as HTML anyway.

Because of this, nearly every major web service routinely suffers from numerous HTML injection flaws; xssed.com, an external site dedicated to tracking publicly reported issues of this type, amassed over 50,000 entries in under two years - and some of the persistent (server-stored) kind turn out to be rather devastating.

The problem clearly demonstrates the inadequateness of same-origin policies as statically bound to a single domain: not all content shown on a particular site should or may be trusted the same, and permitted to do the same. The ability to either isolate, or restrict privileges for portions of data that currently enjoy unconstrained same-origin privileges, would mitigate the impact of HTML parsing problems and developer errors, and also enable new types of applications to be securely built, and is the focus of many experimental security mechanisms proposed for future browsers (as outlined later on).

Life outside same-origin rules

Various flavors of same-origin policies define a set of restrictions for several relatively recent and particularly dangerous operations in modern browsers - DOM access, `XMLHttpRequest`, cookie setting - but as originally envisioned, the web had no security boundaries built in, and no particular limitations were set on what resources pages may interact with, or in what manner. This promiscuous design holds true to this date for many of the core HTML mechanisms, and this degree of openness likely contributed to the overwhelming success of the technology.

This section discusses the types of interaction not subjected to same-origin checks, and the degree of cross-domain interference they may cause.

Navigation and content inclusion across domains

There are numerous mechanisms that permit HTML web pages to include and display remote sub-resources through HTTP `GET` requests without having these operations subjected to a well-defined set of security checks:

- **Simple multimedia markup:** tags such as `<IMG SRC="...">` or `<BGSOUND SRC="...">` permit `GET` requests to be issued to other sites with the intent of retrieving the content to be displayed. The received payload is then displayed on the current page, assuming it conforms to one of the internally recognized formats. In current designs, the data embedded this way remains opaque to JavaScript, however, and cannot be trivially read back (except for occasional bugs).

- **Remote scripts:** `<SCRIPT SRC="...">` tags may be used to issue `GET` requests to arbitrary sites, likewise. A relatively relaxed JavaScript (or E4X XML) parser is then applied to the received content; if the response passes off as something resembling structurally sound JavaScript, this cross-domain payload may then be revealed to other scripts on the current page; one way to achieve this goal is through redefining callback functions or modifying object prototypes; some browsers further help by providing verbose error messages to `onerror` handlers. The possibility of cross-domain data inclusion poses a risk for all sensitive, cookie-authenticated JSON interfaces, and some other document formats not originally meant to be JavaScript, but resembling it in some aspects (e.g., XML, CSV).*Note: quite a few JSON interfaces not intended for cross-domain consumption rely on a somewhat fragile defense: the assumption that certain very specific object serializations (`{ param: "value"}`) or meaningless prefixes (`&&&START&&&`) will not parse via `<SCRIPT SRC="...">`; or that endless loop prefixes such as `while (1)` will prevent interception of the remainder of the data. In most cases, these assumptions are not likely to be future-safe; a better option is to require custom `XMLHttpRequest` headers, or employ a parser-breaking prefix that is unlikely to ever work as long as the basic structure of Java*Script is maintained. One such example is the string of `)]}'`, followed by a newline.

- **Remote stylesheets:** `<LINK REL="stylesheet" HREF="...">` tags may be used in a manner similar to `<SCRIPT>`. The returned data would be subjected to a considerably more rigorous CSS syntax parser. On one hand, the odds of a snippet of non-CSS data passing this validation are low; on the other, the parser does not abort on the first syntax error, and continues parsing the document unconditionally until EOF - so scenarios where some portions of a remote document contain user-controlled strings, followed by sensitive information, are of concern. Once properly parsed, CSS

data may be disclosed to non-same-origin scripts through getComputedStyle or currentStyle properties (the former is W3C-mandated). One potential attack of this type was proposed by Chris Evans; in Internet Explorer, the impact may be greater due to the more relaxed newline parsing rules.

- **Embedded objects and applets:** `<EMBED SRC="...">`, `<OBJECT CODEBASE="...">`, and `<APPLET CODEBASE="...">` tags permit arbitrary resources to be retrieved via `GET` and then supplied as input to browser plugins. The exact effect of this action depends on the plugin to which the resource is routed, a factor entirely controlled by the author of the page. The impact is that content never meant to be interpreted as a plugin-based program may end up being interpreted and executed in a security context associated with the serving host.

- **Document-embedded frames:** `<FRAME>` and `<IFRAME>` elements may be used to create new document rendering containers within the current browser window, and to fetch any target documents via `GET`. These documents would be subject to same-origin checks once loaded.

Note that on all of the aforementioned inclusion schemes other than `<FRAME>` and `<IFRAME>`, any `Content-Type` and `Content-Disposition` HTTP headers returned by the server for the sub-resource are mostly ignored; there is no opportunity to authoritatively instruct the browser about the intended purpose of a served document to prevent having the data parsed as JavaScript, CSS, etc.

In addition to these content inclusion methods, multiple ways exist for pages to initiate full-page transitions to remote content (which causes the target document for such an operation to be replaced with a brand new document in a new security context):

- **Link targets:** the current document, any other named window or frame, or one of special window classes (`_blank`, `_parent`, `_self`, `_top`) may be targeted by `<A HREF="...">` to initiate a regular, `GET`-based page transition. In some browsers, such a link may be also automatically "clicked" by JavaScript with no need for user interaction (for example, using the `click()` method).

- **`Refresh` and `Location` directives:** HTTP `Location` and `Refresh` headers, as well as `<META HTTP-EQUIV="Refresh" VALUE="...">` directives, may be used to trigger a `GET`-based page transition, either immediately or after a predefined time interval.

- **Java**Script DOM access:**Java**Script code may directly access `location.*`, `window.open()`, or `document.URL` to automatically trigger `GET` page transitions, likewise.

- **Form submission:** HTML `<FORM ACTION="...">` tags may be used to submit `POST` and `GET` requests to remote targets. Such transitions may be triggered automatically by JavaScript by calling the `submit()` method. `POST` forms may contain payloads constructed out of form field name-value pairs (both controllable through `<INPUT NAME="..." VALUE="...">` tags), and encoded according to application/x-www-form-urlencoded (`name1=value1&name2=value2...`, with `%nn` encoding of non-URL-safe characters), or to multipart/form-data (a multipart MIME-like format), depending on `ENCTYPE=` parameter.In addition, some browsers permit text/plain to be specified as `ENCTYPE`; in this mode, URL encoding is not applied to `name=value` pairs, allowing almost unconstrained cross-domain `POST` payloads.*Trivia: `POST` payloads are opaque to Java*Script. Without server-side cooperation, or the ability to inspect the originating page, there is no possibility for scripts to inspect the data posted in the request that produced the current page. The property might be relied upon as a crude security mechanism in some specific scenarios, although it does not appear particularly future-safe.

Related tests:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Are verbose `onerror` messages produced for `<SCRIPT>`? | YES | YES | YES | YES | NO | NO | NO | NO | NO | | Are verbose `onerror` messages produced for `<STYLE>`? | NO | NO | NO | NO | NO | NO | NO | NO | NO | | Can links be auto-clicked via `click()`? | YES | YES | YES | NO | NO | NO | YES | NO | NO | | Is `getComputedStyle` supported for CSS? (W3C) | NO | NO | NO | YES | YES | YES | YES | YES | YES | | Is `currentStyle` supported for CSS? (Microsoft) | YES | YES | YES | NO | NO | NO | YES | NO | NO | | Is `ENCTYPE=text/plain` supported on forms | YES | YES | YES | YES | YES | NO | YES | NO | NO |

Note that neither of the aforementioned methods permits any control over HTTP headers. As noted earlier, more permissive mechanisms may be available to plugin-interpreted programs and other non-HTML data, however.

Arbitrary page mashups (UI redressing)

Yet another operation permitted across domains with no specific security checks is the ability to seamlessly merge `<IFRAME>` containers displaying chunks of third-party sites (in their respective security contexts) inside the current document. Although this feature has no security consequences for static content - and in fact, might be desirable - it poses a significant concern with complex web applications where the user is authenticated with cookies: the attacker may cleverly decorate portions of such a third-party UI to make it appear as if they belong to his site instead, and then trick his visitors into interacting with this mashup. If successful, clicks would be directed to the attacked domain, rather than attacker's page - and may result in undesirable and unintentional actions being taken in the context of victim's account.

There are several basic ways to fool users into generating such misrouted clicks:

- **Decoy UI underneath, proper UI made transparent using CSS `opacity` or `filter` attribute:** most browsers permit page authors to set transparency on cross-domain `<IFRAME>` tags. Low opacity may result in the attacked cross-domain UI being barely visible, or not visible at all, with

the browser showing attacker-controlled content placed underneath instead. Any clicks intended to reach attacker's content would still be routed to the invisible third-party UI overlaid on top, however.

- **Decoy UI on top, with a small fragment not covered:** the attacker may also opt for showing the entire UI of the targeted application in a large `<IFRAME>`, but then cover portions of this container with opaque `<DIV>` or `<IFRAME>` elements placed on top (higher CSS `z-index` values). These overlays would be showing his misleading content instead, spare for the single button borrowed from the UI underneath.
- **Keyhole view of the attacked application:** a variant of the previous attack technique is to simply make the `<IFRAME>` very small, and scroll this view to a specific X and Y location where the targeted button is present. Luckily, all current browser no longer permit cross-domain `window.scrollTo()` and `window.scrollBy()` calls - although the attack is still possible if useful HTML anchors on the target page may be repurposed.
- **Race condition attacks:** lastly, the attacker may simply opt for hiding the target UI (as a frame, or as a separate window) underneath his own, and reveal it only miliseconds before the anticipated user click, not giving the victim enough time to notice the switch, or react in any way. Scripts have the ability to track mouse speed and position over the entire document, and close or rearrange windows, but it is still relatively difficult to reliably anticipate the timing of single, casual clicks. Timing solicited clicks (e.g. in action games) is easier, but there is a prerequisite of having an interesting and broadly appealing game to begin with.

In all cases, the attack is challenging to carry out given the deficiencies and incompatibilities of CSS implementations, and the associated difficulty of determining the exact positioning for the targeted UI elements. That said, real-world exploitation is not infeasible. In two of the aforementioned attack scenarios, the fact that that the invisible container may follow mouse pointer on the page makes it somewhat easier to achieve this goal, too.

Also note that the same UI redress possibility applies to `<OBJECT>`, `<EMBED>`, and `<APPLET>` containers, although typically with fewer security implications, given the typical uses of these technologies.

A variant of the attack, relying on a clever manipulation of text field focus, may also be utilized to redirect keystrokes and attempt more complicated types of cross-site interaction.

Mouse-based UI redress attacks gained some prominence in 2008, after Jeremiah Grossman and Robert 'RSnake' Hansen coined the term *clickjacking* and presented the attack to the public. Discussions with browser vendors on possible mitigations are taking place (example), but no definitive solutions are to be expected in the short run. So far, the only freely available product that offers a reasonable degree of protection against the possibility is NoScript (with the recently introduced ClearClick extension). To a much lesser extent, on opt-in defense is available Microsoft Internet Explorer 8, Safari 4, and Chrome 2, through a `X-Frame-Options` header (reference), enabling pages to refuse being rendered in any frames at all (`DENY`), or in non-same-origin ones only (`SAMEORIGIN`).

On the flip side, only a single case of real-world exploitation is publicly known as of this writing.

In absence of browser-side fixes, there are no particularly reliable and non-intrusive ways for applications to prevent attacks; one possibility is to include JavaScript to detect having the page rendered within a cross-domain `<IFRAME>`, and try to break out of it, e.g.:

```
try { if (top.location.hostname != self.location.hostname) throw 1; } catch (e) { top.location.href = self.location.href; }
```

It should be noted that there is no strict guarantee that the update of `top.location` would always work, particularly if dummy setters are defined, or if there are collaborating, attacker-controlled `<IFRAME>` containers performing conflicting location updates through various mechanisms. A more drastic solution would be to also overwrite or hide the current document pending page transition, or to perform `onclick` checks on all UI actions, and deny them from within frames. All of these mechanisms also fail if the user has JavaScript disabled globally, or for the attacked site.

Likewise, because of the features of JavaScript, the following is enough to prevent frame busting in Microsoft Internet Explorer 7:

```
var location = "clobber";
```

Joseph Schorr cleverly pointed out that this behavior may be worked around by creating an `<A HREF="..." TARGET="_top">` HTML tag, and then calling the `click()` on this element; on the other hand, flaws in SECURITY=RESTRICTED frames and the inherent behavior of browser XSS filters render even this variant of limited use; a comprehensive study of these vectors is given in this research paper.

Relevant tests:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Is CSS `opacity` supported ("decoy underneath")? | YES | YES | YES | YES | YES | YES | YES | YES | YES | | Are partly obstructed `IFRAME` containers clickable ("decoy on top")? | YES | YES | YES | YES | YES | YES | YES | YES | YES | | Is cross-domain `scrollBy` scrolling permitted? | NO | NO | NO | NO | NO | NO | NO | NO | n/a | | Is cross-domain anchor-based frame positioning permitted? | YES | YES | YES | YES | YES | YES | YES | YES | n/a | | Is `X-Frame-Options` defense available? | NO | NO | YES | NO | YES | YES | NO | YES | n/a |

Gaps in DOM access control

For compatibility or usability reasons, and sometimes out of simple oversight, certain DOM properties and methods may be invoked across domains without the usual same-origin check carried out elsewhere. These exceptions to DOM security rules include:

- **The ability to look up named third-party windows by their name:** by design, all documents have the ability to obtain handles of all standalone windows and `<IFRAME>` objects they spawn from within JavaScript. Special builtin objects also permit them to look up the handle of the document that embeds them as a sub-resource, if any (`top` and `parent`); and the document that spawned their window (`opener`).On top of this, however, many browsers permit arbitrary other named window to be looked up using `window.open('',<name>)`, regardless of any relation - or lack thereof - between the caller and the target of this operation. This poses a potential security problem (or at least may be an annoyance) when multiple sites are open simultaneously, one of them rogue: although most of user-created windows and tabs are not named, many script-opened windows, IFRAMEs, or "open in new window" link targets, have specific names.*Trivia: `window.name` property, set for top-level windows or `<IFRAME>` containers, is a convenient way to write and read back session-related tokens in absence of cookies. The information stored there is preserved across page transitions, until the window is closed or renamed.*

- **The ability to navigate windows with known handles or names:** subject to certain browser-specific restrictions, browsers may also change the location of windows for which names or JavaScript handles may be obtained. Name-based location changes may be achieved through `window.open(<url>,<name>)` or - outside JavaScript - via `<A HREF="..." TARGET="...">` links (which, as discussed in previous section, in some browsers may be automatically clicked by scripts). Handle-based updates rely on accessing `<win>.location.*` or `document.URL` properties, calling `<win>.location.assign()` or `<win>.location.replace()`, invoking `<win>.history.*` methods, or employing `<win>.document.write`. For windows, the ability to use one of these methods is essentially unrestricted; for frames, it is subject to the descendant policy (DP): the origin of the caller must be the same as the navigated context, one the same with one of its ancestors within the same document window.

- **Assorted coding errors:** a number of other errors and omissions exists, some of which are even depended on to implement legitimate cross-domain communication channels. This category includes missing security checks on `window.opener`, `window.name`, or `window.on*` properties, the ability for parent frames to call functions in child's context, and so forth. DOM Checker is a tool designed to enumerate many of these exceptions and omissions.An important caveat of these mechanisms is that the missing checks might be just as easily used to establish much needed cross-domain communication channels, as they might be abused by third-party sites to inject rogue data into these streams.

- **window.postMessage API:** this new mechanism introduced in several browsers permits two willing windows who have each other's handles to exchange text-based messages across domains as an explicit feature. The receiving party must opt in by registering an appropriate event handler (via `window.addEventListener()`), and has the opportunity to examine `MessageEvent.origin` property to make rudimentary security decisions.

A survey of these exceptions is summarized below:

| **Test description** | **MSIE6** | **MSIE7** | **MSIE8** | **FF2** | **FF3** | **Safari** | **Opera** | **Chrome** | **Android** | |:---|:---|:---|:---|:---|:---|:---|:---|:---|:---|:---| | Can `window.open()` look up unrelated windows? | YES | YES | NO | YES | YES | YES | NO | YES | *YES* | | *Can `frames[]` look up unrelated windows?* | *NO* | *NO* | *NO* | *NO* | *NO* | *NO* | *NO* | *NO* | *NO* | | *Can `<win>.frames[]` navigate third-party IFRAMEs?* | *YES* | *DP* | *DP* | *DP* | *DP* | *DP* | *DP* | *DP* | DP | | Is `<win>.frames[]` iterator permitted? | YES | YES | YES | NO | NO | NO | (NO) | NO | NO | | Can `window.open()` reposition unrelated windows? | YES | YES | NO | YES | YES | YES | NO | YES | *YES* | | *Can `<win>.history.` methods be called on unrelated windows?* | NO | NO | (NO) | YES | YES | YES | NO | YES | *YES* | | *Can `<win>.location.` properties be set on unrelated windows?* | YES | YES | (NO) | YES | YES | YES | NO | YES | *YES* | | *Can `<win>.location.` methods be called on unrelated windows?* | YES | YES | (NO) | YES | YES | YES | NO | YES | *YES* | | *Can `<win>.document.write()` be called on unrelated windows?* | *NO* | *NO* | *NO* | *YES* | *NO* | *NO* | *NO* | *NO* | *NO* | | *Can `TARGET=` links reposition unrelated windows?* | *YES* | *YES* | *NO* | *YES* | *YES* | *YES* | *YES* | *YES* | YES | | Is setting `window.on*` properties possible across domains? | YES (?) | YES (?) | YES (?) | NO | NO | NO | NO | NO | NO | | Is setting `window.opener` possible across domains? | YES | YES | NO | NO | NO | NO | NO | NO | NO | | Is setting `window.name` possible across domains? | YES | YES | NO | NO | NO | NO | NO | NO | NO | | Is calling `frameElements` methods possible across domains? | NO | NO | NO | YES | NO | NO | NO | NO | NO | | Can top-level documents navigate subframes of third-party frames? | YES | YES | NO | YES | YES | YES | NO | YES | YES | | Is `postMessage` API supported? | NO | NO | YES | NO | YES | YES | YES | YES | YES |

* In Chrome, this succeeds only if both tabs share a common renderer process, which limits the scope of possible attacks.

Privacy-related side channels

As a consequence of cross-domain security controls being largely an afterthought, there is no strong compartmentalization and separation of browser-managed resource loading, cache, and metadata management for unrelated, previously visited sites - nor any specific protections that would prevent one site from exploiting these mechanisms to unilaterally and covertly collect fairly detailed information about user's general browsing habits.

Naturally, when the ability for `www.example-bank.com` to find out that their current visitor also frequents `www.example-casino.com` is not mitigated effectively, such a design runs afoul of user's expectations and may be a nuisance. Unfortunately, there is no good method to limit these risks without severely breaking backward compatibility, however.

Aside from coding vulnerabilities such as cross-site script inclusion, some of the most important browsing habit disclosure scenarios include:

- **Reading back CSS `:visited` class on links:** cascading stylesheets support a number of pseudo-classes that may be used by authors to define conditional visual appearance of certain elements. For hyperlinks, these pseudo-classes include `:link` (appearance of an unvisited link), `:hover` (used while mouse hovers over a link), `:active` (used while link is selected), and `:visited` (used on previously visited links).Unfortunately, in conjunction with the previously described getComputedStyle and currentStyle APIs, which are designed to return current, composite CSS data for any given HTML element, this last pseudo-class allows any web site to examine which sites (or site sub-resources) of an arbitrarily large set were visited by the victim, and which were not: if the computed style of a link to `www.example.com` has `:visited` properties applied to it, there is a match.

  *Trivia: even in absence of these APIs, or with Java*Script disabled, somewhat less efficient purely CSS-based enumeration is possible by referencing a unique server-side image via target-specific `:visited` descriptors (https://bugzilla.mozilla.org/show_bug.cgi?id=57351'>more), or detecting document layout changes in other ways.

- **Full-body CSS theft:** as indicated in earlier sections, CSS parsers are generally very strict - but they fail softly: in case of any syntax errors, they do not give up, but rather attempt to locate the next valid declaration and resume parsing from there (this behavior is notably different from JavaScript, which uses a more relaxed parser, but gives up on the first syntax error). This particular well-intentioned property permits a rogue third-party site to include any HTML page, such as `mbox.example-webmail.com`, as a faux stylesheet - and have the parser extract CSS definitions embedded on this page between `<STYLE>` and `</STYLE>` tags only, silently ignoring all the HTML in between.Since many sites use very different inline stylesheets for logged in users and for guests, and quite a few services permit further page customizations to suit users' individual tastes - accessing the `getComputedStyle` or `currentStyle` after such an operation enables the attacker to make helpful observations about victim's habits on targeted sites. A particularly striking example of this behavior is given by Chris Evans in this post.

- **Resource inclusion probes with `onload` and `onerror` checks:** many of the sub-resource loading tags, such as `<IMG>`, `<SCRIPT>`, `<IFRAME>`, `<OBJECT>`, `<EMBED>`, or `<APPLET>`, will invoke `onload` or `onerror` handlers (if defined) to communicate the outcome of an attempt to load the requested URL.Since it is a common practice for various sub-resources on complex web sites to become accessible only if the user is authenticated (returning HTTP 3xx or 4xx codes otherwise), the attacker may carry out rogue attempts to load assorted third-party URLs from within his page, and determine whether the victim is authenticated with cookies on any of the targeted sites.

- **Image size fingerprinting:** a close relative of `onload` and `onerror` probing is the practice of querying `Image.height`, `Image.width`, `getComputedStyle` or `currentStyle` APIs on `<IMG>` containers with no dimensions specified by the page they appear on. A successful load of an authentication-requiring image would result in computed dimensions different from these used for a "broken image" stub.

- **Document structure traversal:** most browsers permit pages to look up third-party named windows or `<IFRAME>` containers across domains. This has two important consequences in the context of user fingerprinting: one is that may be is possible to identify whether certain applications are open at the same time in other windows; the other is that by loading third-party applications in an `<IFRAME>` and trying to look up their sub-frames, if used, often allows the attacker to determine if the user is logged in with a particular site.On top of that, some browsers also leak information across domains by throwing different errors if a property referenced across domains is not found, and different if found, but permission is denied. One such example is the `delete <win>.program_variable` operator.

- **Cache timing:** many resources cached locally by the browser may, when requested, load in a couple milliseconds - whereas fetching them from the server may take a longer while. By timing `onload` events on elements such as `<IMG>` or `<IFRAME>` with carefully chosen target URLs, a rogue page may tell if the requested resource, belonging to a probed site, is already cached - which would indicate prior visits - or not.The probe works only once, as the resources probed this way would be cached for a while as a direct result of testing; but premature retesting could be avoided in a number of ways.

- **General resource timing:** in many web applications, certain pages may take substantially more time to load when the user is logged in, compared to a non-authenticated state; the initial view of a mailbox in a web mail system is usually a very good example. Chris Evans explores this in more detail in his blog post.

- **Pseudo-random number generator probing:** a research paper by Amit Klein explores the idea of reconstructing the state of non-crypto-safe pseudo-random number generators used globally in browsers for purposes such as implementing JavaScript `Math.random()`, or generating `multipart/form-data` MIME boundaries, to uniquely identify users and possibly check for certain events across domains.Since, similarly to libc `rand()`, `Math.random()` is not guaranteed or expected to offer any security, it is important to remember that the output of this PRNG may be predicted or affected in a number of ways, and should never be depended on for security purposes.

Assorted tests related to the aforementioned side channels:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is detection of `:visited` styles possible? YES YES YES YES NO NO NO NO NO Can image sizes be read back via CSS? NO NO NO YES YES YES NO YES YES Can image sizes be read back via `Image` object? YES YES YES YES YES YES YES YES YES Does CSS parser accept HTML documents as stylesheets? YES YES YES YES YES NO YES NO YES Does `onerror` fire on all common HTTP errors? YES YES YES YES YES YES YES YES YES Is `delete <win>.var` probe possible? NO NO NO YES YES NO NO NO NO

Note: Chris Evans and Billy Rios explore many of these vectors in greater detail in their 2008 presentation, https://docs.google.com/Present?docid=dfgb2455_72fkwc2phc'>"Cross-Domain Leakiness".

Various network-related restrictions ## On top of the odd mix of same-origin security policies and one-off exceptions to these rules, there is a set of very specific and narrow connection-related security rules implemented in browsers through the years to address various security flaws. This section provides an overview of these limitations. ### Local network / remote network divide ### The evolution of network security in the recent year resulted in an interesting phenomenon: many home and corporate networks now have very robust external perimeter defenses, filtering most of the incoming traffic in accordance with strict and well-audited access rules. On the flip side, the same networks still generally afford only weak and permissive security controls from within, so any local workstation may deal a considerable amount of damage. Unfortunately for this model, browsers permit attacker-controlled JavaScript or HTML to breach this boundary and take various constrained but still potentially dangerous actions from the network perspective of a local node. Because of this, it seems appropriate to further restrict the ability for such content to interact with any resources not meant to be visible to the outside world. In fact, not doing so already resulted in some otherwise avoidable and scary [real-world attacks] (http://jeremiahgrossman.blogspot.com/2008/04/intranet-hack-targeting-at-2wire-dsl.html). That said, it is not trivial to determine what constitutes a protected asset on an internal network, and what is meant to be visible from the Internet. There are several proposed methods of approximating this set, however; possibilities include blocking access to: * Sites resolving to [RFC 1918](http://www.faqs.org/rfcs/rfc1918.html) address spaces reserved for private use - as these are not intended to be routed publicly, and hence would never point to any meaningful resource on the Internet. * Sites not referenced through [fully-qualified domain names] (http://en.wikipedia.org/wiki/FQDN) - as addresses such as `http://intranet/` have no specific, fixed meaning to general public, but are extensively used on corporate networks. * Address and domain name patterns detected by other heuristics, such as IP ranges local to machine's network interfaces, DNS suffixes defined on proxy configuration exception lists, and so forth. Because none of these methods is entirely bullet-proof or problem-free, as of now, a vast majority of browsers implement no default protection against Internet → intranet fenceposts - although such mechanisms are being planned, tested, or offered optionally in response to previously demonstrated attacks. For example, Microsoft Internet Explorer 7 has a _"Websites in less privileged web content zone can navigate into this zone"_ setting that, if unchecked for _"Local intranet"_, would deny external sites access to a configurable subset of pages. The restriction is disabled by default, however. Relevant tests: | Test description | MSIE6 | MSIE7 | MSIE8 | FF2 | FF3 | Safari | Opera | Chrome | Android | |:----------|:----------|:----------|:----------|:----------|:----------|:----------|:----------|:----------| | Is direct navigation to RFC 1918 IPs possible? | YES | YES | YES | YES | YES | YES | YES | YES | | Is navigation to non-qualified host names possible? | YES | YES | YES | YES | YES | YES | YES | YES | YES | | Is navigation to names that resolve to RFC 1918 ranges possible? | YES | YES |

YES | YES | YES | YES | YES | YES | YES | ### Port access restrictions ### As noted in [earlier sections](http://code.google.com/p/browsersec/wiki/Part1#Uniform_Resource_Locators), URL structure technically permits an arbitrary, non-standard TCP port to be specified for any request. Unfortunately, this permitted attackers to trick browsers into meaningfully interacting with network services that do not really understand HTTP (particularly by abusing `ENCTYPE="text/plain"` forms, as [explained here] (http://www.remote.org/jochen/sec/hfpa/index.html)); these services would misinterpret the data submitted by the browser and could perform undesirable operations, such as accepting and routing SMTP traffic. Just as likely, said services could produce responses that would be in turn misunderstood to the browser, and trigger browser-side security flaws. A particularly interesting example of the latter problem - dubbed _same-site scripting_ - is discussed by Tavis Ormandy in [this BUGTRAQ post](http://seclists.org/bugtraq/2008/Jan/0270.html); another is the tendency for browsers to interpret completely non-HTTP responses as HTTP/0.9, covered [here] (http://resources.enablesecurity.com/resources/the%20extended%20html%20form%20attack%20i Because of this, a rather arbitrary subset of ports belonging to common (and not so common) network services is in modern days blocked for HTTP and some other protocols in most browsers on the market: | Browser | Blocked ports | |:----------|:----------| | **MSIE6,
MSIE7 19 (chargen), 21 (ftp), 25 (smtp), 110 (pop3), 119 (nntp), 143 (imap2) MSIE8 19 (chargen), 21 (ftp), 25 (smtp), 110 (pop3), 119 (nntp), 143 (imap2), 220 (imap3), 993 (ssl imap3) Firefox, Safari,
Opera,
Chrome,
Android 1 (tcpmux), 7 (echo), 9 (discard), 11 (systat), 13 (daytime), 15 (netstat), 17 (qotd), 19 (chargen), 20 (ftp-data),21 (ftp), 22 (ssh), 23 (telnet), 25 (smtp), 37 (time), 42 (name), 43 (nicname), 53 (domain), 77 (priv-rjs), 79 (finger), 87 (ttylink), 95 (supdup), 101 (hostriame), 102 (iso-tsap), 103 (gppitnp), 104 (acr-nema), 109 (pop2), 110 (pop3), 111 (sunrpc), 113 (auth), 115 (sftp), 117 (uccp-path), 119 (nntp), 123 (ntp), 135 (loc-srv), 139 (netbios), 143 (imap2), 179 (bgp), 389 (ldap), 465 (ssl smtp), 512 (exec), 513 (login), 514 (shell), 515 (printer), 526 (tempo), 530 (courier), 531 (chat), 532 (netnews), 540 (uucp), 556 (remotefs), 563 (ssl nntp), 587 (smtp submission), 601 (syslog), 636 (ssl ldap), 993 (ssl imap), 995 (ssl pop3), 2049 (nfs), 4045 (lockd), 6000 (X11) There usually are various protocol-specific exceptions to these rules: for example, `ftp://` URLs are obviously permitted to access port 21, and `nntp://` may reference port 119. A detailed discussion of these exceptions in the prevailing Mozilla implementation is available here.**

URL scheme access rules

The section on URL schemes notes that for certain URL types, browsers implement additional restrictions on what pages may use them and when. Whereas the rationale for doing so for special and dangerous schemes such as `res:` or `view-cache:` is rather obvious, the reasons for restricting two other protocols - most notably `file:` and `javascript:` - are more nuanced.

In the case of `file:`, web sites are generally prevented from navigating to local resources at all. The three explanations given for this decision are as follows:

- Many browsers and browser plugins keep their temporary files and cache data in predictable locations on the disk. The attacker could first plant a HTML file in one these spots during normal browsing activities, and then attempt to open it by invoking a carefully crafted `file:///` URL. As noted earlier, many implementations of same-origin policies are eager to bestow special privileges on local HTML documents (more), and so this led to frequent trouble.
- Users were uncomfortable with random, untrusted sites opening local, sensitive files, directories, or applications within `<IFRAME>` containers, even if the web page embedding them technically had no way to read back the data - a property that a casual user could not verify.
- Lastly, tags such as `<SCRIPT>` or `<LINK REL="stylesheet" HREF="...">` could be used to read back certain constrained formats of local files from the disk, and access the data from within cross-domain scripts. Although no particularly useful and universal attacks of this type were demonstrated, this posed a potential risk.

Browser behavior when handling `file:` URLs in Internet content is summed up in the following table:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Are `<IMG> file:` targets allowed to load? YES YES YES NO NO NO NO NO n/a Are `<SCRIPT> file:` targets allowed to load? YES YES YES NO NO NO NO NO n/a Are `<IFRAME> file:` targets allowed to load? YES YES YES NO NO NO NO NO n/a Are `<EMBED> file:` targets allowed to load? NO NO NO NO NO NO NO NO n/a Are `<APPLET> file:` targets allowed to load? YES YES YES NO NO YES NO YES n/a Are stylesheet `file:` targets allowed to load? YES YES YES NO NO NO NO NO n/a *NOTE: For Microsoft Internet Explorer, the exact behavior for `file:` references is controlled by the "Websites in less privileged web content zone can navigate into this zone" setting for "My computer" zone.* The restrictions applied to `javascript:` are less drastic, and have a different justification. In the past, many HTML sanitizers implemented by various web applications could be easily circumvented by employing seemingly harmless tags such as `<IMG>`, but pointing their `SRC=` parameters to JavaScript pseudo-URLs, instead of HTTP resources - leading to HTML injection and cross-site scripting. To make the life of web developers easier, a majority of browser vendors locked down many of the scenarios where a legitimate use of `javascript:` would be, in their opinion, unlikely:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Are `<IMG> javascript:` targets allowed to run? YES NO NO NO NO NO YES NO NO Are `<SCRIPT> javascript:` targets allowed to run? YES NO NO NO NO NO YES NO NO Are `<IFRAME> javascript:` targets allowed to run? YES YES YES YES YES YES YES YES YES Are `<EMBED> javascript:` targets allowed to run? NO NO NO NO YES NO YES NO n/a Are `<APPLET> javascript:` targets allowed to run? NO NO NO NO YES NO NO NO n/a Are stylesheet `javascript:` targets allowed to run? YES NO NO NO NO NO YES NO NO

Redirection restrictions

Similar to the checks placed on `javascript:` URLs in some HTML tags, HTTP `Location` and HTML `<META HTTP-EQUIV="Refresh" ...>` redirects carry certain additional security restrictions on pseudo-protocols such as `javascript:` or `data:` in most browsers. The reason for this is that it is not clear what security context should be associated with such target URLs. Sites that operate simple open redirectors for the purpose of recording click counts or providing interstitial warnings could fall prey to cross-site scripting attacks more easily if redirects to `javascript:` and other schemes that inherit their execution context from the calling content were permitted.

A survey of current implementations is documented below:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is `Location` redirection to `file:` permitted? NO NO NO NO NO NO NO NO n/a Is `Location` redirection to `javascript:` permitted? NO NO NO NO NO NO NO NO NO Is `Location` redirection to `data:` permitted? n/a n/a n/a NO YES YES YES NO NO Is `Refresh` redirection to `file:` permitted? NO NO NO NO NO NO NO NO n/a Is `Refresh` redirection to `javascript:` permitted? NO NO NO YES NO partly YES YES NO Is `Refresh` redirection to `data:` permitted? n/a n/a n/a YES YES YES YES YES NO Same-origin XMLHttpRequest redirection permitted? YES YES YES YES YES YES YES YES NO Redirection may also fail in a browser-specific manner on some other types of special requests, such as `/favicon.ico` lookups or `XMLHttpRequest`.

International Domain Name checks

The section on international domain names noted that certain additional security restrictions are also imposed on what characters may be used in IDN host names. The reason for these security measures stems from the realization that many Unicode characters are homoglyphs - that is, they look very much like other, different characters from the Unicode alphabet. For example, the following table shows several Cyrillic characters and their Latin counterparts with completely different UTF-8 codes:

Latin a c e i j o p s x y **Cyrillic** a c e i j o p s x y Because of this, with unconstrained IDN support in effect, attacker could easily register www.example.com (Cyrillic character shown in red), and trick his victims into believing they are on the real and original www.example.com site (all Latin). Although "weak" homograph attacks were known before - e.g., www.example.com and www.examp1e.com or www.exarnple.com may look very similar in many typefaces - IDN permitted a wholly new level of domain system abuse.

For a short while, until the first reports pointing out the weakness came in, the registrars apparently assumed that browsers would be capable of detecting such attacks - and browser vendors assumed that it is the job of registrars to properly screen registrations. Even today, there is no particularly good solution to IDN homoglyph attacks available at this time. In general, browsers tend to implement one of the following strategies:

- Not doing anything about the problem, and just displaying Unicode IDN as-is.
- Reverting to Punycode notation when characters in a script not matching user's language settings appear in URLs. This practice is followed by Microsoft Internet Explorer (vendor information), Safari (vendor information), and Chrome. The approach is not bulletproof - as users with certain non-Latin scripts configured may be still easily duped - and tends to cause problems in legitimate uses.
- Reverting to Punycode on all domains with the exception of a whitelisted set, where the registrars are believed to be implementing robust anti-phishing measures. The practice is followed by Firefox (more details). Additionally, as a protection against IDN characters that have no legitimate use in domain names in any script (e.g., www.example-bank.com/evil.fuzzy-bunnies.ch), a short blacklist of characters is also incorporated into the browser - although the mechanism is far from being perfect. Opera takes a similar route (details), and ships with a broader set of whitelisted domains. A general problem with this approach is that it is very difficult to accurately and exhaustively assess actual implementations of phishing countermeasures implemented by hundreds of registrars, or the appearance of various potentially evil characters in hundreds of typefaces - and then keep the list up to date; another major issue is that with Firefox implementation, it rules out any use of IDN in TLDs such as .com.
- Displaying Punycode at all times. This option ensures the purity of traditional all-Latin domain names. On the flip side, the approach penalizes users who interact with legitimate IDN sites on a daily basis, as they have to accurately differentiate between non-human-readable host name strings to spot phishing attempts against said sites.

Experimental solutions to the problem that could potentially offer better security, including color-coding IDN characters in domain names, or employing homoglyph-aware domain name cache to minimize the risk of spoofing against any sites the user regularly visits, were discussed previously, but none of them gained widespread support.

Simultaneous connection limits

For performance reasons, most browsers regularly issue requests simultaneously, by opening multiple TCP connections. To prevent overloading servers with excessive traffic, and to minimize the risk of abuse, the number of connections to the same target is usually capped. The following table captures these limits, as well as default read timeouts for network resources:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Maximum number of same-origin connections 4 4 6 2 6 4 4 6 4 Network read timeout 5 min 5 min 2 min 5 min 10 min 1 min 5 min 5 min 2 min

Third-party cookie rules

Another interesting security control built on top of the existing mechanisms is the concept of restricting third-party cookies. For the privacy reasons http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_cookies'>noted earlier, there appeared to be a demand for a seemingly simple improvement: restricting the ability for any domain other than the top-level one displayed in the URL bar, to set cookies while a page is being visited. This was to prevent third-party content (advertisements, etc) included via `<IMG>`, `<IFRAME>`, `<SCRIPT>`, and similar tags, from setting tracking cookies that could identify the user across unrelated sites relying on the same ad technology.

A setting to disable third-party cookies is available in many browsers, and in several of them, the option is enabled by default. Microsoft Internet Explorer is a particularly interesting case: it rejects third-party cookies with the default "automatic" setting, and refuses to send existing, persistent ones to third-party content (http://msdn.microsoft.com/en-us/library/ms537343(VS.85).aspx#privacy_preference_settings'>"leashing"), but permits sites to override this behavior by declaring a proper, user-friendly intent through http://www.w3.org/P3P/P3FAQ.html'>compact P3P privacy policy headers (a mechanism discussed in more detail http://www.oreillynet.com/pub/a/network/excerpt/p3p/p3p.html'>here and http://www.oreillynet.com/pub/a/network/excerpt/p3p/p3p.html'>here). If a site specifies a privacy policy and the policy implies that personally identifiable information is not collected (e.g., `P3P: CP=NOI NID NOR`), with default security settings, session cookies are permitted to go through regardless of third-party cookie security settings.

The purpose of this design is to force legitimate businesses to make a (hopefully) binding legal statement through this mechanism, so that violations could be prosecuted. Sadly, the approach has the unfortunate property of being a legislative solution to a technical problem, bestowing potential liability at site owners who often simply copy-and-paste `P3P` header examples from the web without understanding their intended meaning; the mechanism also does nothing to stop shady web sites from making arbitrary claims in these HTTP headers and betting on the mechanism never being tested in court - or even simply http://www.interesting-people.org/archives/interesting-people/200112/msg00196.html'>disavowing any responsibility for untrue, self-contradictory, or nonsensical P3P policies.

The question of what constitues "first-party" domains introduces a yet another, incompatible same-origin check, called http://msdn.microsoft.com/en-

us/library/ms537343(VS.85).aspx#first_and_third_party_context'>minimal domains. The idea is that `www1.eu.example.com` and `www2.us.example.com` should be considered first-party, which is not true for all the remaining same-origin logic in other places. Unfortunately, these implementations are generally even more buggy than cookies for country-code TLDs: for example, in Safari, `test1.example.cc` and `test2.example.cc` are *not* the same minimal domain, while in Internet Explorer, `domain1.waw.pl` and `domain2.waw.pl` are.

Although any third-party cookie restrictions are not a sufficient method to prevent cross-domain user tracking, they prove to be rather efficient in disrupting or impacting the security of some legitimate web site features, most notably certain web gadgets and authentication mechanisms.

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Are restrictions on third-party cookies on in default config? YES YES YES NO NO YES NO NO NO Option to change third-party cookie handling? YES YES YES NO YES YES persistent only YES NO Is P3P policy override supported? YES YES YES n/a NO NO n/a NO n/a Does interaction with the IFRAME override cookie blocking? NO NO NO n/a NO YES[*] n/a NO n/a Are third-party cookies permitted within same domain? YES YES YES n/a YES YES n/a YES n/a Behavior of minimal domains in ccTLDs (3 tests) 1/3 FAIL 1/3 FAIL 3/3 PASS n/a 3/3 PASS 1/3 FAIL n/a 3/3 PASS n/a

[*] This includes script-initiated form submissions.

Content handling mechanisms

The task of detecting and handling various file types and encoding schemes is one of the most hairy and broken mechanisms in modern web browsers. This situation stems from the fact that for a longer while, virtually all browser vendors were trying to both ensure backward compatibility with `HTTP/0.9` servers (the protocol included absolutely no metadata describing any of the content returned to clients), and compensate for incorrectly configured `HTTP/1.x` servers that would return HTML documents with nonsensical `Content-Type` values, or unspecified character sets. In fact, having as many content detection hacks as possible would be perceived as a competitive advantage: the user would not care whose fault it was, if `example.com` rendered correctly in Internet Explorer, but not open in Netscape browser - Internet Explorer would be the winner.

As a result, each browser accumulated a unique and very poorly documented set of obscure content sniffing quirks that - because of no pressure on site owners to correct the underlying configuration errors - are now required to keep compatibility with existing content, or at least appear to be risky to remove or tamper with.

Unfortunately, all these design decisions preceded the arrival of complex and sensitive web applications that would host user content - be it baby photos or videos, rich documents, source code files, or even binary blobs of unknown structure (mail attachments). Because of the limitations of same-origin policies, these very applications would critically depend on having the ability to reliably and accurately instruct the browser on how to handle such data, without ever being second-guessed and having what meant to be an image rendered as HTML - and no mechanism to ensure this would be available.

This section includes a quick survey of key file handling properties and implementation differences seen on the market today.

Survey of content sniffing behaviors

The first and only method for web servers to clearly indicate the purpose of a particular hosted resource is through the `Content-Type` response header. This header should contain a standard MIME specification of document type - such as `image/jpeg` or `text/html` - along with some optional information, such as the character set. In theory, this is a simple and bullet-proof mechanism. In practice, not very much so.

The first problem is that - as noted on several occasions already - when loading many types of sub-resources, most notably for `<OBJECT>`, `<EMBED>`, `<APPLET>`, `<SCRIPT>`, `<IMG>`, `<LINK REL="...">`, or `<BGSOUND>` tags, as well as when requesting some plugin-specific, security-relevant data, the recipient would flat out ignore any values in `Content-Type` and `Content-Disposition` headers (or, amusingly, even HTTP status codes). Instead, the mechanism typically employed to interpret the data is as follows:

- General class of the loaded sub-resource is derived from tag type. For example, `<IMG>` narrows the options down to a handful of internally supported image formats; and `<EMBED>` permits only non-native plugins to be invoked. Depending on tag type, a different code path is typically taken, and so it is impossible for `<IMG>` to load a Flash game, or `<EMBED>` to display a JPEG image.
- The exact type of the resource is then decided based on MIME type hints provided in the markup, if supported in this particular case. For example, `<EMBED>` permits a `TYPE=` parameter to be specified to identify the exact plugin to which the data should be routed. Some tags, such as `<IMG>`, offer no provisions to provide any hints as to the exact image format used, however.
- Any remaining ambiguity is then resolved in an implementation- and case-specific manner. For example, if `TYPE=` parameter is missing on `<EMBED>`, server-returned `Content-Type` may be finally examined and compared with the types registered by known plugins. On the other hand, on `<IMG>`, the distinction between JPEG and GIF would be made solely by inspecting the returned payload, rather than interpreting HTTP headers.

This mechanism makes it impossible for any server to opt out from having its responses passed to a variety of unexpected client-side interpreters, if any third-party page decides to do so. In many cases, misrouting the data in this manner is harmless - for example, while it is possible to construct a quasi-valid HTML document that also passes off as an image, and then load it via `<IMG>` tag, there is little or no security risk in allowing such a behavior. Some specific scenarios pose a major hazard, however: one such example is the infamous http://blogs.zdnet.com/security/?p=1619'>GIFAR flaw, where well-formed, user-supplied images could be also interpreted as Java code, and executed in the security context of the serving party.

The other problem is that although `Content-Type` is generally honored for any top-level content displayed in browser windows or within `<IFRAME>` tags, browsers are prone to second-guessing the intent of a serving party, based on factors that could be easily triggered by the attacker. Whenever any user-controlled file that never meant to be interpreted as HTML is nevertheless displayed this way, an obvious security risk arises: any JavaScript embedded therein would execute in the security context of the hosting domain.

The exact logic implemented here is usually contrived and as poorly documented - but based on our current knowledge, could be generalized as:

- If HTTP `Content-Type` header (or other origin-provided MIME type information) is available **and** parses cleanly, it is used as a starting point for further analysis. The syntax for `Content-Type` values is only vaguely outlined in RFC 2045, but generally the value should match a regex of "`[a-z0-9\-]+/[a-z0-9\-]+`" to work properly.Note that protocols such as `javascript:`, `file://`, or `ftp://` do not carry any associated MIME type information, and hence will not satisfy this requirement. Among other things, this property causes the behavior of downloaded files to be potentially very different from that of the same files served over HTTP.
- If `Content-Type` data is not available or did not parse, most browsers would try to guess how to handle the document, based on implementation- and case-specific procedures, such as scanning the first few hundred bytes of a resource, or examining apparent file extension on the end of URL path (or in query parameters), then matching it against system-wide list (`/etc/mailcap`, Windows registry, etc), or a builtin set of rules.Note that due to mechanisms such as `PATH_INFO`, `mod_rewrite`, and other server and application design decisions, the apparent path - used as a content sniffing signal - may often contain bogus, attacker-controlled segments.
- If `Content-Type` matches one of generic values, such as `application/octet-stream`, `application/unknown`, or even `text/plain`, many browsers treat this as a permission to second-guess the value based on the aforementioned signals, and try to come up with something more specific. The rationale for this step is that some badly configured web servers fall back to these types on all returned content.
- If `Content-Type` is valid but not recognized - for example, not handled by the browser internally, not registered by any plugins, and not seen in system registry - some browsers may again attempt to second-guess how to handle the resource, based on a more conservative set of rules.
- For certain `Content-Type` values, browser-specific quirks may also kick in. For example, Microsoft Internet Explorer 6 would try to detect HTML on any `image/png` responses, even if a valid PNG signature is found (this was recently fixed).
- At this point, the content is either routed to the appropriate renderer, or triggers an open / download prompt if no method to internally handle the data could be located. If the appropriate parser does not recognize the payload, or detects errors, it may cause the browser to revert to last-resort content sniffing, however.

  An important caveat is that if `Content-Type` indicates any of XML document varieties, the content may be routed to a general XML parser and interpreted in a manner inconsistent with the apparent `Content-Type` intent. For example, `image/svg+xml` may be rendered as XHTML, depending on top-level or nested XML namespace definitions, despite the fact that `Content-Type` clearly states a different purpose.

As it is probably apparent by now, not much thought or standardization was given to browser behavior in these areas previously. To further complicate work, the documentation available on the web is often outdated, incomplete, or inaccurate (https://developer.mozilla.org/en/How_Mozilla_determines_MIME_Types'>Firefox docs are an example). Following widespread complaints, current http://www.w3.org/html/wg/html5/'>HTML 5 drafts attempt to take at least some content handling considerations into account - although these rules are far from being comprehensive. Likewise, some improvements to specific browser implementations are being gradually introduced (e.g., http://msdn.microsoft.com/en-us/library/cc994329(VS.85).aspx'>image/* behavior changes), while other were resisted (e.g., http://blogs.msdn.com/ie/archive/2005/02/01/364581.aspx'>fixing text/plain logic).

Some of the interesting corner cases of content sniffing behavior are captured below:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is HTML sniffed when no `Content-Type` received? YES YES YES YES YES YES YES YES YES Content sniffing buffer size when no `Content-Type` seen 256 B ∞ ∞ 1 kB 1 kB 1 kB ~130 kB 1 kB ∞ Is HTML sniffed when a non-parseable `Content-Type` value received? NO NO NO YES YES NO YES YES YES Is HTML sniffed on `application/octet-stream` documents? YES YES YES NO NO YES YES NO NO Is HTML sniffed on `application/binary` documents? NO NO NO NO NO NO NO NO NO Is HTML sniffed on `unknown/unknown` (or `application/unknown`) documents? NO NO NO NO NO NO YES NO Is HTML sniffed on MIME types not known to browser? NO NO NO NO NO NO NO NO NO Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL parameters? YES NO NO NO NO NO NO NO NO Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL path? YES YES YES NO NO NO NO NO NO Is HTML sniffed on `text/plain` documents (with or without file extension in URL)? YES YES YES NO NO YES NO NO NO Is HTML sniffed on GIF served as `image/jpeg`? YES YES NO NO NO NO NO NO NO Is HTML sniffed on corrupted images? YES YES NO NO NO NO NO NO NO Content sniffing buffer size for second-

guessing MIME type 256 B 256 B 256 B n/a n/a ∞ n/a n/a n/a May image/svg+xml document contain HTML xmlns payload? (YES) (YES) (YES) YES YES YES YES YES (YES) HTTP error codes ignored when rendering sub-resources? YES YES YES YES YES YES YES YES YES

In addition, the behavior for non-HTML resources is as follows (to test for these, please put sample HTML inside two files with extensions of `.TXT` and `.UNKNOWN`, then attempt to access them through the browser):

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android File type detection for `ftp://` resources content sniffing content sniffing content sniffing content sniffing content sniffing content sniffing extension matching content sniffing n/a File type detection for `file://` resources content sniffing sniffing w/o HTML sniffing w/o HTML content sniffing content sniffing content sniffing content sniffing extension matching n/a

Microsoft Internet Explorer 8 gives an option to override some of its quirky content sniffing logic with a new `X-Content-Type-Options: nosniff` option (http://blogs.msdn.com/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx'>reference). Unfortunately, the feature is somewhat counterintuitive, disabling not only dangerous sniffing scenarios, but also some of the image-related logic; and has no effect on plugin-handled data.

An interesting study of content sniffing signatures is given on http://webblaze.cs.berkeley.edu/2009/content-sniffing/'>this page.

Downloads and Content-Disposition

Browsers automatically present the user with the option to download any documents for which the returned `Content-Type` is:

- Not claimed by any internal feature,
- Not recognized by MIME sniffing or extension matching routines,
- Not handled by any loaded plugins,
- Not associated with a whitelisted external program (such as a media player).

Quite importantly, however, the server may also explicitly instruct the browser not to attempt to display a document inline by employing http://www.ietf.org/rfc/rfc2183.txt'>RFC 2183 `Content-Disposition: attachment` functionality. This forces a download prompt even if one or more of the aforementioned criteria is satisfied - but only for top-level windows and `<IFRAME>` containers.

An explicit use of `Content-Disposition: attachment` as a method of preventing inline rendering of certain documents became a commonly employed *de facto* security feature. The most important property that makes it useful for mitigating the risk of content sniffing is that when included in HTTP headers returned for `XMLHttpRequest`, `<SCRIPT SRC="...">`, or `<IMG SRC="...">` callbacks, it has absolutely no effect on the intended use; but it prevents the attacker from making any direct reference to these callbacks in hope of having them interpreted as HTML.

Closer to its intended use - triggering browser download prompts in response to legitimate and expected user actions - `Content-Disposition` offers fewer benefits, and any reliance on it for security purposes is a controversial practice. Although such a directive makes it possible to return data such as unsanitized `text/html` or `text/plain` without immediate security risks normally associated with such operations, it is not without its problems:

- **Real security gain might be less than expected:** because of the relaxed security rules for `file://` URLs in some browsers - including the ability to access arbitrary sites on the Internet - the benefit of having the user save a file prior to opening it might be illusory: even though the original context is abandoned, the new one is powerful enough to wreak the same havoc on the originating site.Recent versions of Microsoft Internet Explorer mitigate the risk by storing mark-of-the-web and ADS Zone.Identifier tags on all saved content; the same practice is followed by Chrome. These tags are later honored by Internet Explorer, Windows Explorer, and a handful of other Microsoft applications to either restrict the permissions for downloaded files (so that they are treated as if originating from an unspecified Internet site, rather than local disk), or display security warnings and request a confirmation prior to displaying the data. Any benefit of these mechanisms is lost if the data is stored or opened using a third-party browser, or sent to any other application that does not carry out additional checks, however.
- **Loss of MIME metadata may turn harmless files into dangerous ones:** `Content-Type` information is discarded the moment a resource is saved to disk. Unless a careful control is exercised either by the explicit `filename=` field included in `Content-Disposition` headers, or the name derived from apparent URL path, undesired content type promotion may occur (e.g., JPEG becoming an EXE that, to the user, appears to be coming from `www.example-bank.com` or other trusted site). Some, but not all, browsers take measures to mitigate the risk by matching `Content-Type` with file extensions prior to saving files.
- **No consistent rules for escaping cause usability concerns:** certain characters are dangerous in `Content-Disposition` headers, but not necessarily undesirable in local file names (this includes `"` and `;`, or high-bit UTF-8). Unfortunately, there is no reliable approach to encoding non-ASCII values in this header: some browsers support RFC 2047 (`?q?` / `?b?` notation), some support RFC 2231 (a bizarre `*=` syntax), some support stray `%nn` hexadecimal encoding, and some do not support any known escaping scheme at all. As a result, quite a few applications take potentially insecure ways out, or cause problems in non-English speaking countries.

- **Browser bugs further contribute to the misery:** browsers are plagued by implementation flaws even in such a simple mechanism. For example, Opera intermittently ignores `Content-Disposition: attachment` after the initial correctly handled attempt to open a resource, while Microsoft Internet Explorer violates RFC 2183 and RFC 2045 by stopping file name parsing on first `;` character, even within a quoted string (e.g., `Content-Disposition: attachment; filename="hello.exe;world.jpg"` → `"hello.exe"`). Historically, Microsoft Internet Explorer 6, and Safari permitted `Content-Disposition` to be bypassed altogether, too, although these problems appear to be fixed.

Several tests that outline key `Content-Disposition` handling differences are shown below:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is `Content-Disposition` supported correctly? YES YES YES YES YES YES NO YES YES Is Mark-of-the-Web / Zone.Identifier supported? YES YES YES NO NO NO NO write-only NO Types of file name encodings supported `%nn %nn %nn ?b? ?q? *= ?b? ?q? *=` none `*= ?b? ?q? %nn` none Does `filename=test.html` override `Content-Type`? YES YES YES YES YES NO NO NO YES Does `filename=test.exe` override `Content-Type`? YES YES YES YES YES NO NO NO YES Does URL-derived `test.html` filename override `Content-Type`? YES NO YES YES NO NO NO YES Does URL-derived `test.exe` filename override `Content-Type`? YES NO NO YES YES NO NO NO YES Is `;` handled correctly in file names? NO NO NO YES YES YES YES YES YES

Character set handling and detection

When displaying any renderable, text-based content - such as `text/html`, `text/plain`, and many others - browsers are capable of recognizing and interpreting a large number of both common and relatively obscure character sets (http://wiki.whatwg.org/wiki/Web_Encodings'>detailed reference). Some of the most important cases include:

- A basic fixed-width 7-bit `us-ascii` charset (reference). The charset is defined only for character values of `\x00` to `\x7F`, although it is technically transmitted as 8-bit data. High bit values are not permitted; in practice, if they appear in text, their most significant bit may be zeroed upon parsing, or they may be treated as `iso-8859-1` or any other 8-bit encoding.
- An assortment of legacy, fixed-width 8-bit character sets built on top of `us-ascii` by giving a meaning to character codes of `\x80` to `\xFF`. Examples here include `iso-8859-1` (default fallback value for most browsers) and the rest of `iso-8859-*` family, `KOI8-*` family, `windows-*` family, and so forth. Although different glyphs are assigned to the same 8-bit code in each variant, all these character sets are identical from the perspective of document structure parsing.
- A range of non-Unicode variable-width encodings, such as Shift-JIS, EUC-* family, Big5, and so forth. In a majority of these encodings, single bytes in the `\x00` to `\x7F` range are used to represent `us-ascii` values, while high-bit and multibyte values represent more complex non-Latin characters.
- A now-common variable-width 8-bit `utf-8` encoding scheme (reference), where special prefixes of `\x80` to `\xFD` are used on top of `us-ascii` to begin high-bit multibyte sequences of anywhere between 2 and 6 bytes, encoding a full range of Unicode characters.
- A less popular variable-width 16-bit `utf-16` encoding (reference), where single words are used to encode `us-ascii` and the primary planes of Unicode, and word pairs are used to encode supplementary planes. The encoding has little and big endian flavors.
- A somewhat inefficient, fixed-width 32-bit `utf-32` encoding (reference), where every Unicode character is stored as a double word. Again, little and big endian flavors are present.
- An unusual variable-width 7-bit `utf-7` encoding scheme (reference) that permits any Unicode characters to be encoded using `us-ascii` text using a special `+...-` string. Literal `+` and some other values (such as `~` or `\`) must be encoded likewise. Behavior on stray high bit characters may vary, similar to that of `us-ascii`.

There are multiple security considerations for many of these schemes, including:

- Unless properly accounted for, any scheme that may clip high bit values could potentially cause unexpected control characters to appear in unexpected places. For example, in `us-ascii`, a high-bit character of `\xBC` (ź), appearing in user input, may suddenly become `\x3C` (<). This does not happen in modern browsers for HTML parsing, but betting on this behavior being observed everywhere is not a safe assumption to make.
- Unicode-based variable-width `utf-7` and `utf-8` encodings technically make it possible to encode `us-ascii` values using multibyte sequences - for example, `\xC0\xBC` in `utf-8`, or `+ADw-` in `utf-7`, may both decode to `\x3C` (<). Specifications formally do not permit such notation, but not all parsers pay attention to this requirement. Modern browsers tend to reject such syntax for some encodings, but not for others.
- Variable-width decoders may indiscriminately consume a number of bytes following a multibyte sequence prefix, even if not enough data was in place to begin with - potentially causing portions of document structure to disappear. This may easily result in server's and browser's understanding of HTML structure getting dangerously out of sync. With `utf-8`, most browsers avoid over-consumption of non-high-bit values; with `utf-7`, `EUC-JP`, `Big5`, and many other legacy or exotic encodings, this is not always the case.
- All Unicode-based encodings permit certain special codes that function as layout controls to be encoded. Some of these controls override text display direction or positioning (reference). In some uses, permitting such characters to go through might be disruptive.

The pitfalls of specific, mutually negotiated encodings aside, any case where server's understanding of the current character set might be not in sync with that of the browser is a disaster waiting to happen. Since the same string might have very different meanings depending on which decoding procedure is applied to it, the document might be perceived very differently by the generator, and by the renderer. Browsers tend to auto-detect a wide variety of character sets (see: http://msdn.microsoft.com/en-us/library/ms537500(VS.85).aspx'>Internet Explorer list, http://www.mozilla.org/projects/intl/chardet.html'>Firefox list) based on very liberal and undocumented heuristics, historically including even parent character set inheritance (http://www.hardened-php.net/advisory_032007.142.html'>advisory); just as frighteningly, Microsoft Internet Explorer applies character set auto-detection prior to content sniffing.

This behavior makes it inherently unsafe to return any renderable, user-controlled data with no character set explicitly specified. There are two primary ways to explicitly declare a character set for any served content; first of them is the inclusion of a `charset=` attribute in `Content-Type` headers:

```
Content-Type: text/html; charset=utf-8
```

The other option is an equivalent `<META HTTP-EQUIV="...">` directive (supported for HTML only):

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/plain; charset=utf-8">
```

NOTE: Somewhat confusingly, XML `<?xml version="1.0" encoding="UTF-8">` directive does not authoritatively specify the character set for the purpose of rendering XHTML documents - in absence of one of the aforementioned declarations, character set detection may still take place regardless of this tag.

That said, if an appropriate character set declaration is provided, browsers thankfully tend to obey a specified character set value, with several caveats:

- Invalid character set names cause character set detection to kick in. Sadly, there is little or no consistency in how flexibly character set name might be specified, leading to unnecessary mistakes - for example, `iso-8859-2` and `iso8859-2` are both a valid character set to most browsers, and so many developers learned to pay no attention to how they enter the name; but `utf8` is **not** a valid alias for `utf-8`.
- As noted in the section on HTML language, the precedence of `META HTTP-EQUIV` and `Content-Type` character specifications is not well-defined in any specific place - so if the two directives disagree, it is difficult to authoritatively predict the outcome of this conflict in all current and future browsers.
- There are some reports of some exotic non-security glitches present in Microsoft Internet Explorer when only `Content-Type` headers, but not `META HTTP-EQUIV`, are used for HTML documents in certain scripts.
- In some cases, Internet Explorer may choose to ignore `Content-Type` charset if the first characters of the returned payload happen to be a byte order mark associated with a particular character set (e.g., `+/v8` for UTF-7).

Relevant tests:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is `utf-7` character set supported? YES YES YES YES YES YES NO YES NO May `utf-7` be auto-detected in documents? YES YES YES NO NO NO n/a NO n/a `utf-7` sniffing buffer size ∞ ∞ n/a n/a n/a n/a n/a n/a n/a May `utf-7` be inherited by an `<IFRAME>` across domains? NO YES NO NO NO NO n/a NO n/a Does `us-ascii` parsing strip high bits? NO NO NO NO NO NO NO NO NO Behavior of high-bit data in `utf-7` documents keep keep n/a mangle mangle reject n/a reject n/a May 7-bit ASCII may be encoded as `utf-8` on HTML pages? NO NO NO NO NO NO NO NO NO Is 7-bit ASCII consumption possible in `utf-8`? NO NO NO NO NO NO NO NO NO Is 7-bit ASCII consumption possible in `EUC-JP`? NO NO NO YES YES YES NO YES YES Is 7-bit ASCII consumption possible in `Big5`? NO NO NO NO NO YES NO YES YES `Content-Type` header / `HTTP-EQUIV` tag precedence header header header header header header header header header Does the browser fall back to `HTTP-EQUIV` if header charset invalid? NO NO NO YES YES YES NO YES YES

Document caching

HTTP requests are expensive: it takes time to carry out a TCP handshake with a distant host, and to produce or transmit the response (which may span dozens or hundreds of kilobytes even for fairly simple documents). For this reason, having the browser or an intermediate system - such as a traditional, transparent, or reverse proxy - maintain a local copy of at least some of the data can be expected to deliver a considerable performance improvement.

The possibility of document caching is acknowledged in http://www.ietf.org/rfc/rfc1945.txt'>RFC 1945 (the specification for HTTP/1.0). The RFC asserts that user agents and proxies may apply a set of heuristics to decide what default rules to apply to received content, without providing any specific guidance; and also outlines a set of optional, rudimentary caching controls to supplement whichever default behavior is followed by a particular agent:

- `Expires` response header, a method for servers to declare an expiration date, past which the document must be dropped from cache, and a new copy must be requested. There is a relationship between `Expires` and `Date` headers, although it is underspecified: on one hand, the RFC states that if `Expires` value is earlier or equal to `Date`, the content must not be cached at all (leaving the behavior undefined if `Date` header is not present); on the

other, it is not said whether beyond this initial check, `Expires` date should be interpreted according to browser's local clock, or converted to a new deadline by computing an `Expires` - `Date` delta, then adding it to browser's idea of the current date. The latter would account properly for any differences between clock settings on both endpoints.

- `Pragma` request header, with a single value of `no-cache` defined, permitting clients to override intermediate systems to re-issue the request, rather than returning cached data. Any support for `Pragma` is optional from the perspective of standard compliance, however.
- `Pragma` response header, with no specific meaning defined in the RFC itself. In practice, most servers seem to use `no-cache` responses to instruct the browser and intermediate systems not to cache the response, duplicating the backdated `Expires` functionality - although this is not a part of HTTP/1.0 (and as noted, support for `Pragma` directives is optional to begin with).
- `Last-Modified` response header, permitting the server to indicate when, according to its local clock, the resource was last updated; this is expected to reflect file modification date recorded by the file system. The value may help the client make caching decisions locally, but more importantly, is used in conjunction with `If-Modified-Since` to revalidate cache entries.
- `If-Modified-Since` request header, permitting the client to indicate what `Last-Modified` header it had seen on the version of the document already present in browser or proxy cache. If in server's opinion, no modification since `If-Modified-Since` date took place, a null `304 Not Modified` response is returned instead of the requested document - and the client should interpret it as a permission to redisplay the cached document.Note that in HTTP/1.0, there is no obligation for the client to ever attempt `If-Modified-Since` revalidation for any cached content, and no way to explicitly request it; instead, it is expected that the client would employ heuristics to decide if and when to revalidate.

The scheme worked reasonably well in the days when virtually all HTTP content was simple, static, and common for all clients. With the arrival of complex web applications, however, this degree of control quickly proved to be inadequate: in many cases, the only choice an application developer would have is to permit content to be cached by anyone and possibly returned to random strangers, or to disable caching at all, and suffer the associated performance impact. http://www.w3.org/Protocols/rfc2616/rfc2616.txt'>RFC 2616 (the specification for HTTP/1.1) acknowledges many of these problems, and devotes a good portion of the document to establishing ground rules for well-behaved HTTP/1.1 implementations. Key improvements include:

- Sections 13.4, 13.9, and 13.10 spell out which HTTP codes and methods may be implicitly cached, and which ones may not; specifically, only `200`, `203`, `206`, `300`, and `301` responses are cacheable, and only if the method is not `POST`, `PUT`, `DELETE`, or `TRACE`.
- Section 14.9 introduces a new `Cache-Control` header that provides a very fine-grained control of caching strategies for HTTP/1.1 traffic. In particular, caching might be disabled altogether (`no-cache`), only on specific headers (e.g., `no-cache="Set-Cookie"`), or only where it would result in the data being stored on persistent media (`no-store`); caching on intermediate systems might be controlled with `public` and `private` directives; a non-ambiguous maximum time to live might be provided in seconds (`max-age=...`); and `If-Modified-Since` revalidation might be requested for all uses with `must-revalidate`.*Note: the expected behavior of `no-cache` is somewhat contentious, and varies between browsers; most notably, Firefox still caches `no-cache` responses for the purpose of back and forward navigation within a session, as they believe that a literal interpretation of RFC 2616 overrides the intuitive understanding of this directive (reference). They do not cache `no-store` responses within a session, however; and due to the pressure from banks, also have a special case not to reuse `no-cache` pages over HTTPS.*
- Sections 3.11, 14.26, and others introduce `ETag` response header, an opaque identifier expected to be sent by clients in a conditional `If-None-Match` request header later on, to implement a mechanism similar to `Last-Modified` / `If-Modified-Since`, but with a greater degree of flexibility for dynamic resource versioning.*Trivia: as hinted earlier, `ETag` / `If-None-Match`, as well as `Last-Modified` / `If-Modified-Since` header pairs, particularly in conjunction with `Cache-Control: private, max-age=...` directives, may be all abused to store persistent tokens on client side even when HTTP cookies are disabled or restricted. These headers generally work the same as the `Set-Cookie` / `Cookie` pair, despite a different intent.*
- Lastly, section 14.28 introduces `If-Unmodified-Since`, a conditional request header that makes it possible for clients to request a response only if the requested resource is older than a particular date.

Quite notably, specific guidelines for determining TTLs or revalidation rules in absence of explicit directives are still not given in the new specification. Furthermore, for compatibility, HTTP/1.0 directives may still be used (and in some cases must be, as this is the only syntax recognized by legacy caching engines) - and no clear rules for resolving conflicts between HTTP/1.0 and HTTP/1.1 headers, or handling self-contradictory HTTP/1.1 directives (e.g., `Cache-Control: public, no-cache, max-age=100`), are provided - for example, quoting RFCs: *"the result of a request having both an `If-Unmodified-Since` header field and either an `If-None-Match` or an `If-Modified-Since` header fields is undefined by this specification"*. Disk caching strategy for HTTPS is also not clear, leading to subtle differences between implementations (e.g., Firefox 3 requires `Cache-Control: public`, while Internet Explorer will save to disk unless a configuration option is changed).

All these properties make it an extremely good idea to always use explicit, carefully selected, and **precisely matching** HTTP/1.0 and HTTP/1.1 directives on all sensitive HTTP responses.

Relevant tests:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is `Expires` relative to `Date`? NO NO NO YES NO NO YES NO n/a Does invalid `Expires` value stop caching? NO YES YES YES YES YES NO YES n/a Is `Date` needed for `Expires` to work? NO NO NO NO NO NO YES NO n/a

Does invalid `max-age` stop caching? NO NO NO NO YES NO YES NO n/a Does `Cache-Control` override `Expires` in HTTP/1.0? YES NO NO NO NO NO YES NO n/a Does `no-cache` prevail on `Cache-Control` conflicts? YES YES YES YES YES YES NO YES n/a Does `Pragma: no-cache` work? YES YES YES YES YES YES NO YES n/a

NOTE 1: In addition to caching network resources as-is, many modern web browsers also cache rendered page state, so that "back" and "forward" navigation buttons work without having to reload and reinitialize the entire document; and some of them further provide form field caching and autocompletion that persists across sessions. These mechanisms are not subject to traditional HTTP cache controls, but may be at least partly limited by employing the http://www.htmlcodetutorial.com/forms/_INPUT_AUTOCOMPLETE.html'>AUTOCOMPLETE=OFF HTML attribute.

NOTE 2: One of the interesting consequences of browser-side caching is that rogue networks have the ability to "poison" the browser with malicious versions of HTTP resources, and have these copies persist across sessions and networking environments.

NOTE 3: An interesting "by design" glitch reportedly exists in http://blogs.msdn.com/ieinternals/archive/2009/10/02/Internet-Explorer-cannot-download-over-HTTPS-when-no-cache.aspx'>Microsoft Internet Explorer when handling HTTPS downloads with no-cache directives. We were unable to reproduce this so far, however.

Defenses against disruptive scripts

JavaScript and other browser scripting languages make it very easy to carry out annoying, disruptive, or confusing actions, such as hogging the CPU or memory, opening or moving windows faster than the user can respond, preventing the user from navigating away from a page, or displaying modal dialogs in an endless loop; in fact, these actions are so easy that they are sometimes caused unintentionally by site owners.

Historically, most developers would pay little or no attention to this possibility - but as browsers moved toward running more complex and sensitive applications (including web mail, text editors, online games, and so forth), it became less acceptable for one rogue page to trash the entire browser and potentially cause data loss.

As of today, no browser is truly invulnerable to malicious attacks - but most of them try to mitigate the impact of inadvertent site design issues to some extent. This section provides a quick overview of the default restrictions currently in use.

Popup and dialog filtering logic

Virtually all browsers attempt to limit the ability for sites to open new windows (popups), permitting this to take place only in response to `onclick` events within a short timeframe; some also attempt to place restrictions on other similarly disruptive `window.*` methods - although most browsers do fail in one way or another. A survey of current features and implemented limits is shown below:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Is popup blocker present? YES YES YES YES YES YES YES YES YES Are popups permitted on non-`onclick` events? NO NO NO NO NO NO NO YES Maximum delay between click and popup 0 ms 0 ms 0 ms 1 sec 1 sec 0 ms > 5 sec 0 ms 0 ms Maximum number of per-click popups opened 1 1 1 20[*] > 20[*] ∞ ∞ ∞ 1 Is `window.alert()` limited or can be suppressed? NO NO NO NO NO YES YES NO Is `window.print()` limited or can be suppressed? NO NO NO NO NO NO NO n/a Is `window.confirm()` limited or can be suppressed? NO NO NO NO NO YES YES NO Is `window.prompt()` limited or can be suppressed? NO YES YES NO NO NO YES YES NO

[*] Controlled by `dom.popup_maximum` setting in browser configuration.

Window appearance restrictions

A number of restrictions is also placed on `window.open()` features originally meant to make the appearance of browser windows more flexible (http://www.w3schools.com/HTMLDOM/met_win_open.asp'>reference), but in practice had the unfortunate effect of making it easy to spoof system prompts, mimick http://www.squarefree.com/2004/08/01/preventing-browser-ui-spoofing/'>trusted browser chrome, and cause other trouble. The following is a survey of these and related `window.*` limitations:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Are Windows permitted to grab full screen? YES YES YES NO NO NO NO NO n/a Are windows permitted to specify own dimensions? YES YES YES YES YES YES YES YES n/a Are windows permitted to specify screen positioning? YES YES YES YES YES YES NO NO n/a Are windows permitted to fully hide URL bar? YES NO NO YES NO YES NO NO YES Are windows permitted to hide other chrome? YES YES YES YES YES YES NO YES YES Are windows permitted to take focus? YES YES YES NO NO NO NO NO YES Are windows permitted to surrender focus? YES YES YES YES YES YES NO NO NO Are windows permitted to reposition self? YES YES YES YES YES YES NO NO n/a Are windows permitted to close non-script windows? prompt prompt prompt NO NO NO YES NO YES

Execution timeouts and memory limits

Further restrictions are placed on script execution times and memory usage:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Maximum busy loop time 1 sec ∞ ∞ 10 sec 10 sec 10 sec responsive responsive 10 sec Call stack size limit ~2500 ~2000 ~3000 ~1000 ~3000 ~500 ~1000 ~18000 ~500 Heap size limit ∞ ∞ ∞ 16M 16M 16M ∞ ∞

Page transition logic

Lastly, restrictions on `onunload` and `onbeforeunload` may limit the ability for pages to suppress navigation. Historically, this would be permitted by either returning an appropriate code from these handlers, or changing `location.*` properties to counter user's navigation attempt:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Can scripts inhibit page transitions? prompt prompt prompt prompt prompt prompt NO prompt prompt Pages may hijack transitions? YES YES YES NO NO NO NO NO NO

Trivia: Firefox 3 also appears to be exploring the possibility of making the status bar a security feature, by making it impossible for scripts to replace link target URLs on the fly. It is not clear if other browser vendors would share this sentiment.

Protocol-level encryption facilities

HTTPS protocol, as outlined in http://tools.ietf.org/html/rfc2818'>RFC 2818, provides browsers and servers with the ability to establish encrypted TCP connections built on top of the http://en.wikipedia.org/wiki/Transport_Layer_Security'>Transport Layer Security suite, and then exchange regular HTTP traffic within such a cryptographic tunnel in a manner that would resist attempts to intercept or modify the data in transit. To establish endpoint identity and rule out man-in-the-middle attacks, server is required to present a http://en.wikipedia.org/wiki/Public_key_certificate'>public key certificate, which would be then validated against a set of http://en.wikipedia.org/wiki/Root_certificate'>signing certificates corresponding to a handful of trusted commercial entities. This list - typically spanning about 100 certificates - ships with the browser or with the operating system.

To ensure mutual authentication of both parties, the browser may also optionally present an appropriately signed certificate. This is seldom practiced in general applications, because the client may stay anonymous until authenticated on HTTP level through other means; the server needs to be trusted before sending it any HTTP credentials, however.

As can be expected, HTTPS is not without a fair share of problems:

- The assumptions behind the entire HTTPS PKI implementation are that firstly, all the signing authorities can be trusted to safeguard own keys, and thoroughly verify the identity of their paying lower-tier customers, hence establishing a web of trust (*"if Verisign cryptographically attests that the guy holding this certificate is the rightful owner of `www.example.com`, it must be so"*); and secondly, that web browser vendors diligently obtain the initial set of root certificates through a secure channel, and are capable of properly validating SSL connections against this list. Neither of these assumptions fully survived the test of time; in fact, the continued relaxation of validation rules and several minor public blunders led to the introduction of controversial, premium-priced Extended Validation certificates (EV SSL) that are expected to be more trustworthy than "vanilla" ones. The purpose of these certificates is further subverted by the fact that there is no requirement to recognize and treat mixed EV SSL and plain SSL content in a special way, so compromising a regular certificate might be all that is needed to subvert EV SSL sites.
- Another contentious point is that whenever visiting a HTTPS page with a valid certificate, the user is presented with only very weak cues to indicate that the connection offers a degree of privacy and integrity not afforded by non-encrypted traffic - most famously, a closed padlock icon displayed in browser chrome. The adequacy of these subtle and cryptic hints for casual users is hotly debated (1, 2); more visible URL bar signaling in newer browsers is often tied with EV SSL certificates, which potentially somewhat diminishes its value.
- The behavior on invalid certificates (not signed by a trusted entity, expired, not matching the current domain, or suffering from any other malady) is even more interesting. Until recently, most browsers would simply present the user with a short and highly technical information about the nature of a problem, giving the user a choice to navigate to the site at own risk, or abandon the operation. The choice was further complicated by the fact that from the perspective of same-origin checks, there is no difference between a valid and an invalid HTTPS certificate - meaning that a rogue man-in-the-middle version of `https://www.example-bank.com` would, say, receive all cookies and other state data kept by the legitimate one.

  It seemed unreasonable to expect a casual user to make an informed decision here, and so instead, many browsers gradually shifted toward not displaying pages with invalid HTTPS certificates at all, regardless of whether the reason why the certificate does not validate is a trivial or a serious one - and then perhaps giving a well-buried option to override this behavior buried in program settings or at the bottom of an interstitial.This all-or-nothing approach resulted in a paradoxical situation, however: the use of non-validating HTTPS certificates (and hence exposing yourself to nuanced, active man-in-the-middle attacks) is presented by browsers as a problem considerably more serious than the use of open text HTTP communications (and hence exposing the traffic to trivial and unsophisticated passive snooping on TCP level). This practice prevented some sites from taking advantage of the privacy benefits afforded by ad-hoc, MITM-prone cryptography, and again raised some eyebrows.
- Lastly, many types of request and response sequences associated with the use of contemporary web pages can be very likely uniquely fingerprinted based on the timing, direction, and sizes of the exchanged packets alone, as the protocol offers no significant facilities for masking this information

(reference). The information could be further coupled with the knowledge of target IP addresses, and the content of the target site, to achieve a very accurate understanding of user actions at any given time; this somewhat undermines the privacy of HTTPS browsing.

There is also an interesting technical consideration that is not entirely solved in contemporary browsers: HTTP and HTTPS resources may be freely mixed when rendering a document, but doing so in certain configurations may expose the security context associated with HTTPS-served HTML to man-in-the-middle attackers. This is particularly problematic for `<SCRIPT>`, `<LINK REL="stylesheet" ...>`, `<EMBED>`, and `<APPLET>` resources, as they may jump security contexts quite easily. Because of the possibility, many browsers take measures to detect and block at least some mixed content, sometimes breaking legitimate applications in the process.

An interesting recent development is http://lists.w3.org/Archives/Public/www-archive/2009Sep/att-0051/draft-hodges-strict-transport-sec-05.plain.html'>Strict Transport Security, a mechanism that allows websites to opt in for HTTPS-only rendering and strict HTTPS certificate validation through a special HTTP header. Once the associated header - `Strict-Transport-Security` - is seen in a response, the browser will automatically bump all HTTP connection attempts to that site to HTTPS, and will reject invalid HTTPS certificates, with no user recourse. The mechanism offers an important defense against phishing and man-in-the-middle attacks for sensitive sites, but comes with its own set of gotchas - including the fact it requires many existing sites to be redesigned (and SSL content isolated in a separate domain), or that it bears some peripheral denial-of-service risks.

Strict Transport Security is currently supported only by Chrome 4, and optionally through Firefox extensions such as http://noscript.net/'>NoScript.

Several important HTTPS properties and default behaviors are outlined below:

Test description MSIE6 MSIE7 MSIE8 FF2 FF3 Safari Opera Chrome Android Strict Transport Security supported? NO NO NO NO NO NO NO YES NO `Referer` header sent on HTTPS → HTTPS navigation? YES YES YES YES YES YES NO YES YES `Referer` header sent on HTTPS → HTTP navigation? NO NO NO NO NO NO NO NO NO Behavior on invalid certificates prompt interstitial interstitial prompt block prompt prompt interstitial prompt Is EV SSL visually distinguished? NO YES[*] YES NO YES NO YES YES NO Does mixing EV SSL and SSL turn off the EV SSL indicator? n/a NO NO n/a NO n/a NO NO n/a Mixed content behavior on `<IMG>` block block block warn warn permit permit permit permit Mixed content behavior on `<SCRIPT>` block block block warn warn permit permit permit permit Mixed content behavior on stylesheets block block block warn warn permit permit permit permit Mixed content behavior on `<APPLET>` permit permit permit permit permit permit permit permit n/a Mixed content behavior on `<EMBED>` permit permit permit permit permit permit permit permit n/a Mixed content behavior on `<IFRAME>` block block block warn warn permit permit permit permit Do wildcard certificates match multiple host name segments? NO NO NO YES YES NO NO NO NO

[*] On Windows XP, this is enabled only when http://support.microsoft.com/kb/931125'>KB931125 is installed, and browser's phishing filter functionality is enabled.

Trivia: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/attachments/20080714/07ea5534/attachment.txt'>KEYGEN tags are an obscure and largely undocumented feature supported by all browsers with the exception of Microsoft Internet Explorer. These tags permit the server to challenge the client to generate a cryptographic key, send it to server for signing, and store a signed response in user's key chain. This mechanism provides a quasi-convenient method to establish future client credentials in the context of HTTPS traffic.

(http://code.google.com/p/browsersec/wiki/Part3'>Continue to experimental and legacy mechanisms...)