Browser Security Handbook

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

Table of Contents

        - Local network / remote network divide
        - Port access restrictions
        - URL scheme access rules
        - Redirection restrictions
        - International Domain Name checks
        - Simultaneous connection limits
    - Third-party cookie rules
    - Content handling mechanisms
        - Survey of content sniffing behaviors
        - Downloads and Content-Disposition
        - Character set handling and detection
        - Document caching
    - Defenses against disruptive scripts
        - Popup and dialog filtering logic
        - Window appearance restrictions
        - Execution timeouts and memory limits
        - Page transition logic
    - Protocol-level encryption facilities

Part 3

- *← Back to browser security features*

- Experimental and legacy security mechanisms
    - HTTP authentication
    - Name look-ahead and content prefetching
    - Password managers
    - Microsoft Internet Explorer zone model
    - Microsoft Internet Explorer frame restrictions
    - HTML5 sandboxed frames
    - HTML5 storage, cache, and worker experiments
    - Microsoft Internet Explorer XSS filtering
    - Script restriction frameworks
    - Secure JSON parsing
    - Origin headers
    - Mozilla content security policies
- Open browser engineering issues

Introduction

Hello, and welcome to the *Browser Security Handbook*!

This document is meant to provide web application developers, browser engineers, and information security researchers with a one-stop reference to key security properties of contemporary web browsers. Insufficient understanding of these often poorly-documented characteristics is a major contributing factor to the prevalence of several classes of security vulnerabilities.

Although all browsers implement roughly the same set of baseline features, there is relatively little standardization - or conformance to standards - when it comes to many of the less apparent implementation details. Furthermore, vendors routinely introduce proprietary tweaks or improvements that may interfere with existing features in non-obvious ways, and seldom provide a detailed discussion of potential problems.

The current version of this document is based on the following versions of web browsers:

Browser	Version	Test date	Usage*	Notes
Microsoft Internet Explorer 6	6.0.2900.5512	Feb 2, 2009	16%	
Microsoft Internet Explorer 7	7.0.5730.11	Dec 11, 2008	11%	
Microsoft Internet Explorer 8	8.0.6001.18702	Sep 7, 2010	28%	
Mozilla Firefox 2	2.0.0.18	Nov 28, 2008	1%	
Mozilla Firefox 3	3.6.8	Sep 7, 2010	22%	
Apple Safari	4.0	Jun 10, 2009	5%	
Opera	9.62	Nov 18, 2008	2%	
Google Chrome	7.0.503.0	Sep 7, 2010	8%	
Android embedded browser	SDK 1.5 R3	Oct 3, 2009	n/a	

* Approximate browser usage data based on public Net Applications estimates for August 2010.

Disclaimers and typographical conventions

Please note that although we tried to make this document as accurate as possible, some errors might have slipped through. Use this document only as an initial reference, and independently verify any characteristics you wish to depend upon. Test cases for properties featured in this document are freely available for download.

The document attempts to capture the risks and security considerations present for general populace of users accessing the web with default browser settings in place. Although occasionally noted, the degree of flexibility offered through non-standard settings is by itself not a subject of this comparative study.

Through the document, red color is used to bring attention to browser properties that seem particularly tricky or unexpected, and need to be carefully accounted for in server-side implementations. Whenever status quo appears to bear no significant security consequences and is well-understood, but a particular browser implementation takes additional steps to protect application developers, we use green color to denote this, likewise. Rest assured, neither of these color codes implies that a particular browser is less or more secure than its counterparts.

Acknowledgments

Browser Security Handbook would not be possible without the ideas and assistance from the following contributors:

- Filipe Almeida
- Brian Eaton
- Chris Evans
- Drew Hintz
- Nick Kralevich
- Marko Martin
- Tavis Ormandy
- Wladimir Palant
- David Ross
- Marius Schilder
- Parisa Tabriz
- Julien Tinnes

- Berend-Jan Wever
- Mike Wiacek

The document builds on top of previous security research by Adam Barth, Collin Jackson, Amit Klein, Jesse Ruderman, and many other security experts who painstakingly dissected browser internals for the past few years.

Basic concepts behind web browsers

This section provides a review of core standards and technologies behind current browsers, and their security-relevant properties. No specific attention is given to features implemented explicitly for security purposes; these are discussed in later in the document.

Uniform Resource Locators

All web resources are addressed with the use of uniform resource identifiers. Being able to properly parse the format, and make certain assumptions about the data present therein, is of significance to many server-side security mechanisms.

The abstract syntax for URIs is described in RFC 3986. The document defines a basic hierarchical URI structure, defining a white list of unreserved characters that may appear in URIs as-is as element names, with no particular significance assigned (`0-9 A-Z a-z - . _ ~`), spelling out reserved characters that have special meanings and can be used in some places only in their desired function (`: / ? # [ ] @ ! $ & ' ( ) * + , ; =`), and establishing a hexadecimal percent-denoted encoding (`%nn`) for everything outside these sets (including the stray `%` character itself).

Some additional mechanisms are laid out in RFC 1738, which defines URI syntax within the scope of HTTP, FTP, NNTP, Gopher, and several other specific protocols. Together, these RFCs define the following syntax for common Internet resources (the compliance with a generic naming strategy is denoted by the `//` prefix):

```
scheme://[login[:password]@](host_name|host_address)[:port][/hierarchical/path/to/resource[?search_string][#fragment_id]]
```

Since the presence of a scheme is the key differentiator between relative references permitted in documents for usability reasons, and fully-qualified URLs, and since `:` itself has other uses later in the URL, the set of characters permitted for scheme name must be narrow and clearly defined (`0-9 A-Z a-z + - .`) so that all implementations may make the distinction accurately.

On top of the aforementioned documents, a W3C draft RFC 1630 and a non-HTTP RFC 2368 de facto outline some additional concepts, such as the exact HTTP search string syntax (`param1=val1[&param2=val2&...]`), or the ability to use the `+` sign as a shorthand notation for spaces (the character itself does not function in this capacity elsewhere in the URL, which is somewhat counterintuitive).

Although a broad range of reserved characters is defined as delimiters in generic URL syntax, only a subset is given a clear role in HTTP addresses at any point; the function of `[`, `]`, `!`, `$`, `'`, `(`, `)`, `*`, `;`, or `,` is not explicitly defined anywhere, but the characters are sometimes used to implement esoteric parameter passing conventions in oddball web application frameworks. The RFC itself sometimes implies that characters with no specific function within the scheme should be treated as regular, non-reserved ASCII characters; and elsewhere, suggests they retain a special meaning - in both cases creating ambiguities.

The standards that specify the overall URL syntax are fairly laid back - for example, they permit IP addresses such as `74.125.19.99` to be written in completely unnecessary and ambiguous ways such as `74.0x7d.023.99` (mixing decimal, octal, and hexadecimal notation) or `74.8196963` (24 bits coalesced). To add insult to injury, on top of this, browsers deviate from these standards in random ways, for example accepting URLs with technically illegal characters, and then trying to escape them automatically, or passing them as-is to underlying implementations - such as the DNS resolver, which itself then rejects or passes through such queries in a completely OS-specific manner.

A particularly interesting example of URL parsing inconsistencies is the two following URLs. The first one resolves to a different host in Firefox, and to a different one in most other browsers; the second one behaves uniquely in Internet Explorer instead:

```
http://example.com\@coredump.cx/
http://example.com;.coredump.cx/
```

Below is a more detailed review of the key differences that often need to be accounted for:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Characters ignored in front of URL schemes	\x01-\x20	\x01-\x20	\x01-\x20	\t \r \n \x20	\t \r \n \x20	\x20	\t \r \n \x0B \x0C \xA0	\x00-\x20	\x20
Non-standard characters permitted in URL scheme names (excluding `0-9 A-Z a-z + - .`)	\t \r \n	\t \r \n	\t \r \n	\t \r \n	\t \r \n	none	\r \n +UTF8	\0 \t \r \n	none
Non-standard characters kept as-is, with no escaping, in URL query strings (excluding `0-9 A-Z a-z - . _ ~ : / ? # [ ] @ ! $ & ' ( ) * + , ; =`)*	" < > \ ^ ` { | } \x7F	" < > \ ^ ` { | } \x7F	" < > \ ^ ` { | } \x7F	\ ^ { | }	\ ^ { | }	^ { | }	^ { | } \x7F	" \ ^ ` { | }	n/a
Non-standard characters fully ignored in host names	\t \r \n	\t \r \n \xAD	\t \r \n \xAD	\t \r \n \xAD	\t \r \n \xAD	\xAD	\x0A-\x0D \xA0 \xAD	\t \r \n \xAD	none
Types of partial or broken URLs auto-corrected to fully qualified ones	//y \\y	//y \\y	//y \\y	//y x:///y x:// `[y]`	//y x:///y x:// `[y]`	//y \\y x:/y x:///y	//y \\y x:// `[y]`	//y \\y x:///y	//y \\y

Is fragment ID (hash) encoded by applying RFC-mandated URL escaping rules?	NO	NO	NO	PARTLY	PARTLY	YES	NO	NO	YES
Are non-reserved `%nn` sequences in URL path decoded in address bar?	NO	YES	YES	NO	YES	NO	YES	YES	n/a
Are non-reserved `%nn` sequences in URL path decoded in `location.href`?	NO	YES	YES	NO	NO	YES	YES	YES	YES
Are non-reserved `%nn` sequences in URL path decoded in actual HTTP requests sent?	NO	YES	YES	NO	NO	NO	NO	YES	NO
Characters rejected in URL login or password (excluding `/ # ; ? : % @`)	\x00 \	\x00 \	\x00 \	none	none	\x00-\x20 " < > [\ ] ^ ` { | } \x7f-\xff	\x00	\x00 \x01\	\x00-\x20 " < > [\ ] ^ ` { | } \x7f-\xff
URL authentication data splitting behavior with multiple `@` characters	leftmost	leftmost	leftmost	rightmost	rightmost	leftmost	rightmost	rightmost	leftmost

[*] *Interestingly, Firefox 3.5 takes a safer but non-RFC-compliant approach of encoding stray ' characters as %27 in URLs, in addition to the usual escaping rules.*

NOTE: As an anti-phishing mechanism, additional restrictions on the use of `login` and `password` fields in URLs are imposed by many browsers; see the section on HTTP authentication later on.

Please note that when links are embedded within HTML documents, HTML entity decoding takes place before the link is parsed. Because of this, if a tab is ignored in URL schemes, a link such as `javascript&#09;:alert(1)` may be accepted and executed as JavaScript, just as `javascript<TAB>:alert(1)` would be. Furthermore, certain characters, such as `\x00` in Internet Explorer, or `\x08` in Firefox, may be ignored by HTML parsers if used in certain locations, even though they are not treated differently by URL-handling code itself.

Unicode in URLs

Much like several related technologies used for web content transport, URLs do not have any particular character set defined by relevant RFCs;RFC 3986 ambiguously states: *"In local or regional contexts and with improving technology, users might benefit from being able to use a wider range of characters; such use is not defined by this specification."* The same dismissive approach is taken in HTTP header specification.

As a result, in the context of web URLs, any high-bit user input may and should be escaped as `%nn` sequences, but there is no specific guidance provided on how to transcode user input in system's native code page when talking to other parties that may not share this code page. On a system that uses UTF-8 and receives a URL containing Unicode `ą` in the path, this corresponds to a sequence of `0xC4 0x85` natively; however, when sent to a server that uses `ISO-8859-2`, the correct value sent should be `0xB1` (or alternatively, additional information about client-specific encoding should be included to make the conversion possible on server side). In practice, most browsers deal with this by sending UTF-8 data by default on any text entered in the URL bar by hand, and using page encoding on all followed links.

Another limitation of URLs traces back to DNS as such; RFC 1035 permits only characters `A-Z a-z 0-9 -` in DNS labels, with period (`.`) used as a delimiter; some resolver implementations further permit underscore (`_`) to appear in DNS names, violating the standard, but other characters are almost universally banned. With the growth of the web, the need to accommodate non-Latin alphabets in host names was perceived by multiple parties - and the use of `%nn` encoding was not an option, because `%` as such was not on the list.

To solve this, RFC 3490 lays out a rather contrived encoding scheme that permitted Unicode data to be stored in DNS labels, and RFC 3492outlines a specific implementation within DNS labels - commonly referred to as Punycode - that follows the notation of `xn--[US-ASCII part]-[encoded Unicode data]`. Any Punycode-aware browser faced with non US-ASCII data in a host name is expected to transform it to this notation first, and then perform a traditional DNS lookup for the encoded string.

Putting these two methods together, the following transformation is expected to be made internally by the browser:

> `http://www.ręczniki.pl/?ręcznik=1 → http://www.xn--rczniki-98a.pl/?r%C4%99cznik=1`

Key security-relevant differences in high-bit URL handling are outlined below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Request URL path encoding when following plain links	UTF-8	UTF-8	UTF-8	page encoding	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8

Request URL query string encoding when following plain links	page encoding, no escaping	page encoding, no escaping	page encoding, no escaping	page encoding	page encoding	page encoding	page encoding	page encoding	page encoding
Request URL path encoding for `XMLHttpRequest` calls	page encoding	page encoding	page encoding	page encoding	page encoding	page encoding	page encoding	page encoding	page encoding
Request URL query string encoding for `XMLHttpRequest`calls	page encoding, no escaping	page encoding, no escaping	page encoding, no escaping	page encoding	page encoding	mangled	page encoding	mangled	mangled
Request URL path encoding for manually entered URLs	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8	UTF-8
Request URL query string encoding for manually entered URLs	transcoded to 7 bit	transcoded to 7-bit	transcoded to 7-bit	UTF-8	UTF-8	UTF-8	stripped to ?	UTF-8	UTF-8
Raw Unicode in host names auto-converted to Punycode?	NO	YES	YES	YES	YES	YES	YES	YES	NO
Is percent-escaped UTF-8 in host names auto-converted to Punycode?	(NO)	YES	YES	NO	on retry	YES	YES	YES	(NO)
URL bar Unicode display method for host names	(Punycode)	Unicode	Unicode	Unicode	Unicode	Unicode	Unicode	Unicode	(Punycode)
URL bar Unicode display method outside host names	Unicode	Unicode	Unicode	`%nn`	Unicode	Unicode	as ?	Unicode	n/a

NOTE 1: Firefox generally uses UTF-8 to encode most URLs, but will send characters supported by ISO-8859-1 using that codepage.

NOTE 2: As an anti-phishing mechanism, additional restrictions on the use of some or all Unicode characters in certain top-level domains are imposed by many browsers; see domain name restrictions chapter later on.

True URL schemes

URL schemes are used to indicate what general protocol needs to be used to retrieve the data, and how the information needs to be processed for display. Schemes may also be tied to specific default values of some URL fields - such as a TCP port - or specific non-standard URL syntax parsing rules (the latter is usually denoted by the absence of a `//` string after scheme identifier).

Back in 1994, RFC 1738 laid out several URL schemes in the context of web browsing, some of which are not handled natively by browsers, or have fallen into disuse. As the web matured, multiple new open and proprietary schemes appeared with little or no consistency, and the set of protocols supported by each browser began to diverge. RFC 2718 attempted to specify some ground rules for the creation of new protocols, and RFC 4395mandated that new schemes be registered with IANA (their list is available here). In practice, however, few parties could be bothered to follow this route.

To broadly survey the capabilities of modern browsers, it makes sense to divide URL schemes into a group natively supported by the software, and another group supported by a plethora of plugins, extensions, or third-party programs. The first list is relatively short:

Scheme name	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
HTTP (RFC 2616)	YES	YES	YES	YES	YES	YES	YES	YES	YES
HTTPS (RFC 2818)	YES	YES	YES	YES	YES	YES	YES	YES	YES
SHTTP (RFC 2660)	as HTTP	as HTTP	as HTTP	NO	NO	NO	NO	NO	NO
FTP (RFC 1738)	YES	YES	YES	YES	YES	YES	YES	YES	NO

file (RFC 1738)	YES	YES	YES	YES (local)	YES (local)	YES (local)	YES	YES	NO
Gopher (RFC 4266)	NO	NO	NO	YES	YES	NO	defunct?	NO	NO
news (draft RFC)	NO	NO	NO	NO	NO	NO	YES	NO	NO

These protocols may be used to deliver natively rendered content that is interpreted and executed using the security rules implemented within the browsers.

The other list, third-party protocols routed to other applications, depends quite heavily on system configuration. The set of protocols and their handlers is usually maintained in a separate system-wide registry. Browsers may whitelist some of them by default (executing external programs without prompting), or blacklist some (preventing their use altogether); the default action is often to display a mildly confusing prompt. Most common protocols in this family include:

```
acrobat                         - Acrobat Reader
callto                          - some instant messengers, IP phones
daap / itpc / itms / ...        - Apple iTunes
FirefoxURL                      - Mozilla Firefox
hcp                             - Microsoft Windows Help subsystem
ldap                            - address book functionality
mailto                          - various mail agents
mmst / mmsu / msbd / rtsp / ... - streaming media of all sorts
mso-offdap                      - Microsoft Office
news / snews / nntp             - various news clients
outlook / stssync               - Microsoft Outlook
rlogin / telnet / tn3270        - telnet client
shell                           - Windows Explorer
sip                             - various IP phone software
```

New handlers might be registered in OS- and application-specific ways, for example by registering new `HKCR\Protocols\Handlers` keys in Windows registry, or adding `network.protocol-handler.*` settings in Firefox.

As a rule, the latter set of protocols is not honored within the renderer when referencing document elements such as images, script or applet sources, and so forth; they do work, however, as `<IFRAME>` and link targets, and will launch a separate program as needed. These programs may sometimes integrate seamlessly with the renderer, but the rules by which they render content and impose security restrictions are generally unrelated to the browser as such.

Historically, the ability to plant such third-party schemes in links proved to be a significant exploitation vector, as poorly written, vulnerable external programs often register protocol handlers without user's knowledge or consent; as such, it is prudent to reject unexpected schemes where possible, and exercise caution when introducing new ones on client side (including observing safe parameter passing conventions).

Pseudo URL schemes

In addition to the aforementioned "true" URL schemes, modern browsers support a large number of pseudo-schemes used to implement various advanced features, such as encapsulating encoded documents within URLs, providing legacy scripting features, or giving access to internal browser information and data views.

Encapsulating schemes are of interest to any link-handling applications, as these methods usually impose specific non-standard content parsing or rendering modes on top of existing resources specified in the later part of the URL. The underlying content is retrieved using HTTP, looked up locally (e.g., `file:///`), or obtained using other generic method - and, depending on how it's then handled, may execute in the security context associated with the origin of this data. For example, the following URL:

```
jar:http://www.example.com/archive.jar!/resource.html
```

...will be retrieved over HTTP from `http://www.example.com/archive.jar`. Because of the encapsulating protocol, the browser will then attempt to interpret the obtained file as a standard Sun Java ZIP archive (`JAR`), and extract, then display `/resource.html` from within that archive, in the context of `example.com`.

Common encapsulating schemes are shown in the table below.

Scheme name	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
feed (RSS, draft spec)	NO	NO	NO	NO	NO	YES	NO	NO	NO
hcp, its, mhtml, mk, ms-help, ms-its, ms-itss (Windows help archive parsing)	YES	YES	YES	NO	NO	NO	NO	NO	NO
jar (Java archive parsing)	NO	NO	NO	YES	YES	NO	NO	NO	NO
view-cache, wyciwyg (cached page views)	NO	NO	NO	YES	YES	NO	NO	YES	NO

view-source (page source views)	NO	NO	NO	YES	YES	NO	NO	YES	NO

In addition to encapsulating schemes enumerated above, there are various schemes used for accessing browser-specific internal features unrelated to web content. These pseudo-protocols include `about:` (used to access static info pages, errors, cache statistics, configuration pages, and more), `moz-icon:` (used to access file icons), `chrome:`, `chrome-resource:`, `chromewebdata:`, `resource:`, `res:`, and `rdf:` (all used to reference built-in resources of the browser, often rendered with elevated privileges). There is little or no standardization or proper documentation for these mechanisms, but as a general rule, web content is not permitted to directly reference any sensitive data. Permitting them to go through on trusted pages may serve as an attack vector in case of browser-side vulnerabilities, however.

Finally, several pseudo-schemes exist specifically to enable scripting or URL-contained data rendering in the security context inherited from the caller, without actually referencing any additional external or internal content. It is particularly unsafe to output attacker-controlled URLs of this type on pages that may contain any sensitive content. Known schemes of this type include:

Scheme name	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
data (in-place documents, RFC 2397)	NO	NO	PARTIAL	YES	YES	YES	YES	YES	YES
javascript (web scripting)	YES	YES	YES	YES	YES	YES	YES	YES	YES
vbscript (Microsoft proprietary scripting)	YES	YES	YES	NO	NO	NO	NO	NO	NO

NOTE: Historically, numerous aliases for these schemes were also present; `livescript` and `mocha` schemes were supported by Netscape Navigator and other early browsers as aliases for JavaScript; `local` worked in some browsers as a nickname for `file`; etc. This is not witnessed anymore.

Hypertext Transfer Protocol

The core protocol used to request and annotate much of web traffic is called the Hypertext Transfer Protocol. This text-based communication method originated as a very simple, underspecified design drafted by Tim Berners-Lee, dubbed HTTP/0.9 (see W3C archive) - these days no longer used by web browsers, but recognized by some servers. It then evolved into a fairly complex, and still somewhat underspecified HTTP/1.1, as described in RFC 2616, whilst maintaining some superficial compatibility with the original idea.

Every HTTP request opens with a single-line description of a content access method (`GET` meant for requesting basic content, and `POST` meant for submitting state-changing data to servers - along with plethora of more specialized options typically not used by web browsers under normal circumstances). In HTTP/1.0 and up, this is then followed by protocol version specification - and the opening line itself is followed by zero or more additional `field: value` headers, each occupying their own line. These headers specify all sorts of meta-data, from target host name (so that a single machine may host multiple web sites), to information about client-supported MIME types, cache parameters, the site from which a particular request originated (`Referer`), and so forth. Headers are terminated with a single empty line, followed by any optional payload data being sent to the server if specified by a `Content-Length` header.

One example of an HTTP request might be:

```
POST /fuzzy_bunnies/bunny_dispenser.php HTTP/1.1
Host: www.fuzzybunnies.com
User-Agent: Bunny-Browser/1.7
Content-Type: text/plain
Content-Length: 12
Referer: http://www.fuzzybunnies.com/main.html

HELLO SERVER
```

The server responds in a similar manner, returning a numerical status code, spoken protocol version, and similarly formatted metadata headers followed by actual content requested, if available:

```
HTTP/1.1 200 OK
Server: Bunny-Server/0.9.2
Content-Type: text/plain
Connection: close

HELLO CLIENT
```

Originally, every connection would be one-shot: after a request is sent, and response received, the session is terminated, and a new connection needs to be established. Since the need to carry out a complete TCP/IP handshake for every request imposed a performance penalty, newer specifications introduced the concept of keep-alive connections, negotiated with a particular request header that is then acknowledged by the server.

This, in conjunction with the fact that HTTP supports proxying and content caching on interim systems managed by content providers, ISPs, and individual subscribers, made it particularly important for all parties involved in an HTTP transaction to have exactly the same idea of where a request starts, where it ends, and what it is related to. Unfortunately, the protocol itself is highly ambiguous and has a potential for redundancy, which leads to multiple problems and differences between how servers, clients, and proxies may interpret responses:

- Like many other text protocols of that time, early takes on HTTP made little or no effort to mandate a strict adherence to a particular understanding of what a text-based format really is, or how certain "intuitive" field values must be structured. Because of this, implementations would recognize, and often handle in incompatible ways, technically malformed inputs - such as incorrect newline characters (lone `CR`, lone `LF`, `LF CR`), `NUL` or other disruptive control characters in text, incorrect number of whitespaces in field delimiters, and so forth; to various implementations, `Head\0er: Value` may appear as `Head`, `Head: Value`, `Header: Value`, or `Head\0er: Value`. In later versions, as outlined in RFC 2616 section 19.3 ("Tolerant Applications"), the standard explicitly recommends, but does not require, lax parsing of certain fields and invalid values. One of the most striking examples of compatibility kludges is Firefox prtime.c function used to parse HTTP `Date` fields, which shows a stunning complexity behind what should be a remarkably simple task.

- No particular high bit character set is defined for HTTP headers, and high bit characters are allowed by HTTP/1.0 with no further qualification, then technically disallowed by HTTP/1.1, unless encoded in accordance with RFC 2047. In practice, there are legitimate reasons for such characters to appear in certain HTTP fields (e.g., `Cookie`, `Content-Disposition` filenames), and most implementations do not support RFC 2047 in all these places, or find support for it incompatible with other RFCs (such as the specifications for `Cookie` headers again). This resulted in some implementations interpreting HTTP data as UTF-8, and some using single-byte interpretations native to low-level OS string handling facilities.

- The behavior when some headers critical to the correct understanding of an HTTP request are duplicate or contradictory is not well defined; as such, various clients will give precedence to different occurrences of the same parameter within HTTP headers (e.g., duplicate `Content-Type`), or assign various weights to conflicting information (say, `Content-Length` not matching payload length). In other cases, the precedence might be defined, but not intuitive - for example, RFC 2616 section 5.2 says that absolute request URI data takes precedence over `Host` headers.

- When new features that change the meaning of requests were introduced in HTTP/1.1 standard, no strict prohibition against recognizing them in requests or responses marked as HTTP/1.0 was made. As a result, the understanding of HTTP/1.0 traffic may differ significantly between legacy agents, such as some commercial web proxies, and HTTP/1.1 applications such as contemporary browsers (e.g., `Connection: keep-alive`,`Transfer-Encoding: chunked`, `Accept-Encoding: ...`).

Many specific areas, such as caching behavior, have their own sections later in this document. Below is a survey of general security-relevant differences in HTTP protocol implementations:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Header-less (HTTP/0.9) responses supported?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Lone CR (`0x0D`) treated as a header line separator?	YES	YES	YES	NO	NO	NO	YES	YES	NO
`Content-Length` header value overrides actual content length?	NO	YES	YES	NO	NO	YES	YES	NO	YES
First HTTP header of the same name takes precedence?	YES	YES	YES	NO	NO	YES	NO	YES	NO
First field value in a HTTP header takes precedence?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is `Referer` header sent on HTTPS → HTTPS navigation?	YES	YES	YES	YES	YES	YES	NO	YES	YES
Is `Referer` header sent on HTTPS → HTTP navigation?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is `Referer` header sent on HTTP → HTTPS → HTTP redirection?	YES	YES	YES	YES	YES	YES	YES	NO	YES
Is `Referer` header sent on pseudo-protocol → HTTP navigation?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is fragment ID included in `Referer` on normal requests?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is fragment ID included in `Referer` on `XMLHttpRequest`?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Response body on invalid 30x redirect shown to user?	NO	NO	NO	YES	YES	NO	YES	YES	NO
High-bit character handling in HTTP cookies	transcoded to 7 bit	transcoded to 7 bit	transcoded to 7 bit	mangled	mangled	UTF-8	UTF-8	UTF-8	UTF-8

Are `quoted-string` values supported for HTTP cookies?	NO	NO	NO	YES	YES	NO	YES	NO	YES

NOTE 1: `Referer` will always indicate the site from which the navigation originated, regardless of any 30x redirects in between. If it is desirable to hide the original URL from the destination site, JavaScript pseudo-protocol hops, or `Refresh` redirection, needs to be used.

NOTE 2: `Refresh` header tokenization in MSIE occurs in a very unexpected manner, making it impossible to navigate to URLs that contain any literal `;` characters in them, unless the parameter is enclosed in additional quotes. The tokenization also historically permitted cross-site scripting through URLs such as:

```
http://example.com;URL=javascript:alert(1)
```

Unlike in all other browsers, older versions of Internet Explorer would interpret this as two `URL=` directives, with the latter taking precedence:

```
Refresh: 0; URL=http://example.com;URL=javascript:alert(1)
```

Hypertext Markup Language

Hypertext Markup Language, the primary document format rendered by modern web browsers, has its roots with Standard Generalized Markup Language, a standard for machine-readable documents. The initial HTML draft provided a very limited syntax intended strictly to define various functional parts of the document. With the rapid development of web browsers, this basic technology got extended very rapidly and with little oversight to provide additional features related to visual presentation, scripting, and various perplexing and proprietary bells and whistles. Perhaps more interestingly, the format was also extended to provide the ability to embed other, non-HTTP multimedia content on pages, nest HTML documents within frames, and submit complex data structures and client-supplied files.

The mess eventually led to a post-factum compromise standard dubbed HTML 3.2. The outcome of this explosive growth was a format needlessly hard to parse, and combining unique quirks, weird limitations, and deeply intertwined visual style and document structure information - and so ever since, W3C and WHATWG focused on making HTML a clean, strict, and well-defined language, a goal at least approximated with HTML 4 andXHTML (a variant of HTML that strictly conforms to XML syntax rules), as well as the ongoing work on HTML 5.

This day, the four prevailing HTML document rendering implementations are:

- Trident (MSHTML) - used in MSIE6, MSIE7, MSIE8,
- Gecko - used in Firefox and derivates,
- WebKit - used by Safari, Chrome, Android,
- Presto - used in Opera.

The ability for various applications to accurately understand HTML document structure, as it would be seen by a browser, is an important security challenge. The serial nature of the HTML blends together code (JavaScript, Flash, Java applets) and the actual data to be displayed - making it easy for attackers to smuggle dangerous directives along with useful layout information in any external content. Knowing exactly what is being rendered is often crucial to site security (see this article for a broader discussion of the threat).

Sadly, for compatibility reasons, parsers operating in non-XML mode tend to be generally lax and feature proprietary, incompatible, poorly documented recovery modes that make it very difficult for any platform to anticipate how a third-party HTML document - or portion thereof - would be interpreted. Any of the following grossly malformed examples may be interpreted as a scripting directive by some, but usually not all, renderers:

```
01: <B <SCRIPT>alert(1)</SCRIPT>>
02: <B="<SCRIPT>alert(1)</SCRIPT>">
03: <IMG SRC=`javascript:alert(1)`>
04: <S[0x00]CRIPT>alert(1)</S[0x00]CRIPT>
05: <A """><IMG SRC="javascript:alert(1)">
06: <IMG onmouseover =alert(1)>
07: <A/HREF="javascript:alert(1)">
08: <!-- Hello -- world > <SCRIPT>alert(1)</SCRIPT> -->
09: <IMG ALT="><SCRIPT>alert(1)</SCRIPT>"(EOF)
10: <![><IMG ALT="]><SCRIPT>alert(1)</SCRIPT>">
```

Cross-site scripting aside, another interesting property of HTML is that it permits certain HTTP directives to be encoded within HTML itself, using the following format:

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

Not all `HTTP-EQUIV` directives are meaningful - for example, the determination of `Content-Type`, `Content-Length`, `Location`, or `Content-Disposition` had already been made by the time HTML parsing begins - but some values seen may be set this way. The strategy for resolving HTTP - HTML conflicts is not outlined in W3C standards - but in practice, valid HTTP headers take precedence over `HTTP-EQUIV`; on the other hand, `HTTP-EQUIV` takes precedence over unrecognized HTTP header values. `HTTP-EQUIV` tags will also take precedence when the content is moved to non-HTTP media, such as saved to local disk.

Key security-relevant differences between HTML parsing modes in the aforementioned engines are shown below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android

Parser resets on nested HTML tags (<FOO <BAR...)?	NO	NO	NO	YES	YES	YES	YES	YES	YES
Recursive recovery with nested tags (both FOO and BAR interpreted)?	(NO)	(NO)	(NO)	YES	YES	YES	NO	YES	YES
Parser resets out on invalid tag names (<FOO="<BAR...)?	NO	NO	NO	YES	YES	YES	NO	YES	YES
Trace-back on missing tag closure (<FOO BAR="><BAZ>" (EOF))?	YES	YES	YES	NO	NO	NO	YES	NO	NO
Trace-back on missing parameter closure (<FOO BAR="><BAZ> (EOF))?	NO	NO	NO	YES	YES	NO	YES	NO	NO
SGML-style comment parsing permitted in strict mode (-- and > may appear separately)?	NO	NO	NO	YES	YES	NO	NO	NO	NO
CDATA blocks supported in plain HTML documents?	NO	NO	NO	YES	YES	NO	YES	NO	NO
!-type tags are parsed in a non-HTML manner (<!FOO BAR="-->"... breaks)?	NO	NO	NO	YES	YES	NO	YES	NO	NO
Characters accepted as tag name / parameter separators (excluding \t \r \n \x20)	\x0B \x0C /	\x0B \x0C /	NO	/	/	\x0B \x0C	\x0B \x0C \xA0	\x0B \x0C	\x0B \x0C
Characters ignored between parameter name, equals sign, and value (excluding \t \r \n)	\0 \x0B \x0C \x20	\0 \x0B \x0C \x20	\0 \x0B \x0C \x20	\x20	\x20	\0 \x0B \x0C \x20 /	\x20 \xA0	\0 \x0B \x0C \x20/	\0 \x0B \x0C \x20/
Characters accepted in lieu of quotes for HTML parameters (excluding ")	' `	' `	' `	'	'	'	'	'	'
Characters accepted in tag names (excluding A-Z / ? !)	\0 %	\0 %	\0 %	none	none	none	\0	none	none

NOTE: to add insult to injury, special HTML handling rules seem to be sometimes applied to specific sections of a document; for example, \x00 character is ignored by Internet Explorer, and \x08 by Firefox, in certain HTML contexts, but not in others; most notably, they are ignored in HTML tag parameter values in respective browsers.

HTML entity encoding

HTML features a special encoding scheme called HTML entities. The purpose of this scheme is to make it possible to safely render certain reserved HTML characters (e.g., < > &) within documents, as well as to carry high bit characters safely over 7-bit media. The scheme nominally permits three types of notation:

- One of predefined, named entities, in the format of &<name>; - for example < for <, > for >, → for →, etc,
- Decimal entities, &#<nn>;, with a number corresponding to the desired Unicode character value - for example < for <, → for →,
- Hexadecimal entities, &#x<nn>;, likewise - for example < for <, → for →.

In every browser, HTML entities are decoded only in parameter values and stray text between tags. Entities have no effect on how the general structure of a document is understood, and no special meaning in sections such as <SCRIPT>. The ability to understand and parse the syntax is still critical to properly understanding the value of a particular HTML parameter, however. For example, as hinted in one of the earlier sections, <A HREF="javascript	:alert(1)"> may need to be parsed as an absolute reference to javascript<TAB>:alert(1), as opposed to a link to something called javascript& with a local URL hash string part of #09;alert(1).

Unfortunately, various browsers follow different parsing rules to these HTML entity notations; all rendering engines recognize entities with no proper; terminator, and all permit entities with excessively long, zero-padded notation, but with various thresholds:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Maximum length of a correctly terminated decimal entity	7	7	7	∞	∞	8*	∞	8*	8*
Maximum length of an incorrectly terminated decimal entity	7	7	7	∞	∞	8*	∞	8*	8*
Maximum length of a correctly terminated hex entity	6	6	6	∞	∞	8*	∞	8*	8*
Maximum length of an incorrectly terminated hex entity	0	0	0	∞	∞	8*	∞	8*	8*
Characters permitted in entity names (excluding `A-Z a-z 0-9`)	none	none	none	`-.`	`-.`	none	none	none	none

* Entries one byte longer than this limit still get parsed, but incorrectly; for example, `&#000000065;` becomes a sequence of three characters,`\x06 5 ;`. Two characters and more do not get parsed at all - `&#0000000065;` is displayed literally).

An interesting piece of trivia is that, as per HTML entity encoding requirements, links such as:

```
http://example.com/?p1=v1&p2=v2
```

Should be technically always encoded in HTML parameters (but not in JavaScript code) as:

```
<a href="http://example.com/?p1=v1&amp;p2=v2">Click here</a>
```

In practice, however, the convention is almost never followed by web developers, and browsers compensate for it by treating invalid HTML entities as literal `&`-containing strings.

Document Object Model

As the web matured and the concept of client-side programming gained traction, the need arose for HTML document to be programatically accessible and modifiable on the fly in response to user actions. One technology, Document Object Model (also referred to as "dynamic HTML"), emerged as the prevailing method for accomplishing this task.

In the context of web browsers, Document Object Model is simply an object-based representation of HTML document layout, and by a natural extension much of the browser internals, in a hierarchical structure with various read and write properties, and callable methods. To illustrate, to access the value of the first `<INPUT>` tag in a document through DOM, the following JavaScript code could be used:

```
document.getElementsByTagName('INPUT')[0].value
```

Document Object Model also permits third-party documents to be referenced, subject to certain security checks discussed later on; for example, the following code accesses the `<INPUT>` tag in the second `<IFRAME>` on a current page:

```
document.getElementsByTagName('IFRAME')[1].contentDocument.getElementsByTagName('INPUT')[0].value
```

DOM object hierarchy - as seen by programmers - begins with an implicit "root" object, sometimes named `defaultView` or `global`; all the scripts running on a page have `defaultView` as their default scope. This root object has the following members:

- Various properties and methods relating to current document-specific window or frame (reference). Properties include window dimensions, status bar text, references to parent, top-level, and owner (opener) `defaultView` objects. Methods permit scripts to resize, move, change focus, or decor of current windows, display certain UI widgets, or - oddly enough - set up JavaScript timers to execute code after a certain "idle" interval.
- All the current script's global variables and functions.
- `defaultView.document` (reference) - a top-level object for the actual, proper *document* object model. This hierarchy represents all of the document elements, along with their specific methods and properties, and document-wide object lookup functions (e.g., `getElementById`,`getElementsByTagName`, etc). An assortment of browser-specific, proprietary APIs is usually present on top of the common functionality (e.g.,document.execCommand in Internet Explorer).
- `defaultView.location` (reference) - a simple object describing document's current location, both as an unparsed string, and split into parsed segments; provides methods and property setters that allow scripts to navigate away to other sites.
- `defaultView.history` (reference) - another simple object offering three methods to enable pages to navigate back to previously visited pages.

- `defaultView.screen` (reference) - yet another modestly sized object with a bunch of mostly read-only properties describing properties of the current display device, including pixel and DPI resolution, and so forth.

- `defaultView.navigator` (reference) - an object containing read-only properties such as browser make and version, operating platform, or plugin configuration.

- `defaultView.window` - a self-referential entry that points back to the root object; this is the name by which the `defaultView` object is most commonly known. In general, `screen`, `navigator`, and `window` DOM hierarchies combined usually contain enough information to very accurately fingerprint any given machine.

With the exception of cross-domain, cross-document access permissions outlined in later chapters, the operation of DOM bears relatively little significance to site security. It is, however, worth noting that DOM methods are wrappers providing access to highly implementation-specific internal data structures that may or may not obey the usual rules of JavaScript language, may haphazardly switch between bounded and ASCIZ string representations, and so forth. Because of this, multiple seemingly inexplicable oddities and quirks plague DOM data structures in every browsers - and some of these may interfere with client-side security mechanisms. Several such quirks are documented below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is `window` the same object as `window.window`?	NO	NO	NO	YES	YES	YES	YES	YES	YES
Is `document.URL` writable?	NO	YES	YES	NO	NO	NO	NO	NO	NO
Can builtin DOM objects be clobbered?	overwrite	overwrite	overwrite	shadowing	shadowing	shadowing	overwrite	overwrite	shadowing
Does `getElementsByName` look up by `ID=` values as well?	YES	YES	YES	NO	NO	NO	YES	NO	NO
Are `.innerHTML` assignments truncated at `NUL`?	YES	YES	NO	NO	NO	NO	YES	NO	NO
Are `location.*` assignments truncated at `NUL`?	YES	YES	YES	YES	YES	YES	YES	NO	NO

Trivia: traversing `document.*` is a possible approach to look up specific input or output fields when modifying complex pages from within scripts, but it is not a very practical one; because of this, most scripts resort to `document.getElementById()` method to uniquely identify elements regardless of their current location on the page. Although `ID=` parameters for tags within HTML documents are expected to be unique, there is no such guarantee. Currently, all major browsers seem to give the first occurrence a priority.

Browser-side Javascript

JavaScript is a relatively simple but rich, object-based imperative scripting language tightly integrated with HTML, and supported by all contemporary web browsers. Authors claim that the language has roots with Scheme and Self programming languages, although it is often characterized in a more down-to-earth way as resembling a crossover of C/C++ and Visual Basic. Confusingly, except for the common C syntax, it shares relatively few unique features with Java; the naming is simply a marketing gimmick devised by Netscape and Sun in the 90s.

The language is a creation of Netscape engineers, originally marketed under the name of Mocha, then Livescript, and finally rebranded to JavaScript; Microsoft embraced the concept shortly thereafter, under the name of JScript. The language, as usual with web technologies, got standardized only post factum, under a yet another name - ECMAScript. Additional, sometimes overlapping efforts to bring new features and extensions of the language - for example, ECMAScript 4, JavaScript versions 1.6-1.8, native XML objects, etc - are taking place, although none of these concepts managed to win broad acceptance and following.

Browser-side JavaScript is invoked from within HTML documents in four primary ways:

1. Standalone `<SCRIPT>` tags that enclose code blocks,
2. Event handlers tied to HTML tags (e.g. `onmouseover="..."`),
3. Stylesheet `expression(...)` blocks that permit JavaScript syntax in some browsers,
4. Special URL schemes specified as targets for certain resources or actions (`javascript:...`) - in HTML and in stylesheets.

Note that the first option does not always work when such a string is dynamically added by running JavaScript to an existing document. That said, any of the remaining options might be used as a comparable substitute.

Regardless of their source (`<SCRIPT SRC="...">`), remote scripts always execute in the security context of the document they are attached to. Once called, JavaScript has full access to the current DOM, and limited access to DOMs of other windows; it may also further invoke new JavaScript by calling `eval()`, configuring timers (`setTimeout(...)` and `setInterval(...)`), or producing JavaScript-invoking HTML. JavaScript may also configure self to launch when its objects are interacted with by third-party JavaScript code, by configuring watches, setters, or getters, or cross contexts by calling same-origin functions belonging to other documents.

JavaScript enjoys very limited input and output capabilities. Interacting with same-origin document data and drawing CANVASes, displaying`window.*` pop-up dialogs, and writing script console errors aside, there are relatively few I/O facilities available. File operations or access to other persistent storage is generally not possible, although some experimental features are being introduced and quickly scrapped. Scripts may send and receive data from the originating server using the XMLHttpRequest extension,

which permits scripts to send and read arbitrary payloads; and may also automatically send requests and read back properly formatted responses by issuing `<SCRIPT SRC="...">`, or `<LINK REL="Stylesheet" HREF="...">`, even across domains. Lastly, JavaScript may send information to third-party servers by spawning objects such as `<IMG>`, `<IFRAME>`, `<OBJECT>`, `<APPLET>`, or by triggering page transitions, and encoding information in the URL that would be automatically requested by the browser immediately thereafter - but it may not read back the returned data. Some side channels related to browser caching mechanisms and lax DOM security checks also exist, providing additional side channels between cooperating parties.

Some of the other security-relevant characteristics of JavaScript environments in modern browsers include:

- **Dynamic, runtime code interpretation with no strict code caching rules**. Any code snippets located in-line with HTML tags would be interpreted and executed, and JavaScript itself has a possibility to either directly evaluate strings as JavaScript code (`eval(...)`), or to produce new HTML that in turn may contain more JavaScript (`.innerHTML` and `.outerHTML` properties, `document.write()`, event handlers). The behavior of code that attempts to modify own container while executing (through DOM) is not well-defined, and varies between browsers.

- **Somewhat inconsistent exception support**. Although within the language itself, exception-throwing conditions are well defined, it is not so when interacting with DOM structures. Depending on the circumstances, some DOM operations may fail silently (with writes or reads ignored), return various magic responses (`undefined`, `null`, `object inaccessible`), throw a non-standardized exception, or even abort execution unconditionally.

- **Somewhat inconsistent, but modifiable builtins and prototypes**. The behavior of many language constructs might be redefined by programs within the current context by modifying object setters, getters, or overwriting prototypes. Because of this, reliance on any specific code - for example a security check - executing in a predictable manner in the vicinity of a potentially malicious payload is risky. Notably, however, not all builtin objects may be trivially tampered with - for example, `Array` prototype setters may be modifiable, but `XML` not so. There is no fully-fledged operator overloading in JavaScript 1.x.

- **Typically synchronous execution**. Within a single document, browser-side JavaScript usually executes synchronously and in a single thread, and asynchronous timer events are not permitted to fire until the scripting engine enters idle state. No strict guarantees of synchronous execution exist, however, and multi-process rendering of Chrome or MSIE8 may permit cross-domain access to occur more asynchronously.

- **Global function lookups not dependent on ordering or execution flow**. Functions may be invoked in a block of code before they are defined, and are indexed in a pass prior to execution. Because of this, reliance on top-level no-return statements such as `while (1);` to prevent subsequent code from being interpreted might be risky.

- **Endless loops that terminate**. As a security feature, when the script executes for too long, the user is prompted to abort execution. This merely causes current execution to be stopped, but does not prevent scripts from being re-entered.

- **Broad, quickly evolving syntax**. For example, E4X extensions in Firefox well-structured XHTML or XML to be interpreted as a JavaScript object, also executing any nested JavaScript initializers within such a block. This makes it very difficult to assume that a particular format of data would not constitute valid JavaScript syntax in a future-proof manner.

Inline script blocks embedded within HTML are somewhat tricky to sanitize if permitted to contain user-controlled strings, because, much like`<TEXTAREA>`, `<STYLE>`, and several other HTML tags, they follow a counterintuitive CDATA-style parsing: a literal sequence of `</SCRIPT>` ends the script block regardless of its location within the JavaScript syntax as such. For example, the following code block would be ended prematurely, and lead to unauthorized, additional JavaScript block being executed:

```
<SCRIPT>
var user_string = 'Hello world</SCRIPT><SCRIPT>alert(1)</SCRIPT>';
</SCRIPT>
```

Strictly speaking, HTML4 standard specifies that any `</...` sequence may be used to break out of non-HTML blocks such as `<SCRIPT>`(reference); in practice, this suit is not followed by browsers, and a literal `</SCRIPT>` string is required instead. This property is not necessarily safe to rely on, however.

Various differences between browser implementations are outlined below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is setter and getter support present?	NO	NO	NO	YES	YES	YES	YES	YES	YES
Is access to prototypes via `__proto__` possible?	NO	NO	NO	YES	YES	YES	NO	YES	YES
Is it possible to alias `eval()` function?	YES	YES	YES	PARTLY	PARTLY	YES	PARTLY	YES	YES
Are watches on objects supported?	NO	NO	NO	YES	YES	NO	NO	NO	NO
May currently executing code blocks modify self?	read-only	read-only	read-only	NO	NO	NO	YES	NO	NO

Is E4X extension supported?	NO	NO	NO	YES	YES	NO	NO	NO	NO
Is `charset=` honored on `<SCRIPT SRC="...">`?	YES	YES	YES	YES	YES	YES	NO	YES	YES
Do Unicode newlines (U+2028, U+2029) break lines in JavaScript?	NO	NO	NO	YES	YES	YES	YES	YES	YES

Trivia: JavaScript programs can programatically initiate or intercept a variety of events, such as mouse and keyboard input within a window. This can interfere with other security mechanisms, such as "protected" `<INPUT TYPE=FILE ...>` fields, or non-same-origin HTML frames, and little or no consideration was given to these interactions at design time, resulting in a number of implementation problems in modern browsers.

Javascript character encoding

To permit handling of strings that otherwise contain restricted literal text (such as `</SCRIPT>`, `"`, `'`, `CL` or `LF`, other non-printable characters), JavaScript offers five character encoding strategies:

1. Three-digit, zero-padded, 8-bit octal numerical representations, C-style (`"test"` → `"t\145st"`),
2. Two-digit, zero-padded, 8-bit hexadecimal numerical representations, with C-style `\x` prefix (`"test"` → `"t\x65st"`),
3. Four-digit, zero-padded, 16-bit hexadecimal numerical Unicode representations, with `\u` prefix (`"test"` → `"t\u0065st"`),
4. Raw `\` prefix before a literal (`"test"` → `"t\est"`),
5. Special C-style `\` shorthand notation for certain control characters (such as `\n` or `\t`).

The fourth scheme is most space-efficient, although generally error-prone; for example, a failure to escape `\` as `\\` may lead to a string of `\" + alert(1);"` being converted to `\\" + alert(1)`, and getting interpreted incorrectly; it also partly collides with the remaining `\`-prefixed escaping schemes, and is not compatible with C syntax.

The parsing of these schemes is uniform across all browsers if fewer than the expected number of input digits is seen: the value is interpreted correctly, and no subsequent text is consumed (`"\1test"` is accepted and becomes `"\001test"`).

HTML entity encoding has no special meaning within HTML-embedded `<SCRIPT>` blocks.

Amusingly, although all the four aforementioned schemes are supported in JavaScript proper, a proposed informational standard for JavaScript Object Notation (JSON), a transport-oriented serialization scheme for JavaScript objects and arrays (RFC 4627), technically permits only the `\u`notation and a seemingly arbitrary subset of `\` control character shorthand codes (options 3 and 5 on the list above). In practice, since JSON objects are almost always simply passed to `eval(...)` in client-side JavaScript code to convert them back to native data structures, the limitation is not enforced in a vast majority of uses. Because of the advice provided in the RFC, some non-native `parseJSON` implementations (such as thecurrent example reference implementation from json.org) may not handle traditional `\x` escape codes properly, however. It is also not certain what approach would be followed by browsers in the currently discussed native, non-executing parsers proposed for performance and security reasons.

Unlike in some C implementations, stray multi-line string literals are not permitted by JavaScript, but lone `\` at the end of a line may be used to break long lines in a seamless manner.

Other document scripting languages

VBScript is a language envisioned by Microsoft as a general-purpose "hosted" scripting environment, incorporated within Windows as such, Microsoft IIS, and supported by Microsoft Internet Explorer via `vbscript:` URL scheme, and through `<SCRIPT>` tags. As far as client-side scripting is considered, the language did not seem to offer any significant advantages over the competing JavaScript technology, and appeared to lag in several areas.

Presumably because of this, the technology never won broad browser support outside MSIE (with all current versions being able to execute it), and is very seldom relied upon. Where permitted to go through, it should be expected to have roughly the same security consequences as JavaScript. The attack surface and security controls within VBScript are poorly studied in comparison to that alternative, however.

Cascading stylesheets

As the initial HTML designs evolved to include a growing body of eye candy, the language ended up mixing very specific visual presentation cues with content classification directives; for example, a similar inline syntax would denote that the following piece of text is an element of a numbered list, or that it needs to be shown in 72 pt Comic Sans font. This notation, although convenient to use on new pages, made it very difficult to automatically adjust appearance of a document without altering its structure (for example to account for mobile devices, printable views, or accessibility requirements), or to accurately preserve document structure when moving the data to non-HTML media or new site layouts.

Cascading Style Sheets is a simple concept that corrects this problem, and also provides a much more uniform and featured set of tools to alter the visual appearance of any portion of the document, far surpassing the original set of kludgy HTML tag attributes. A stylesheet outlines visual rendering rules for various functional types of document elements, such as lists, tables, links, or quotations, using a separate block of data with a relatively simple syntax. Although the idea of style sheets as such predates HTML, the current design used for the web stabilized in the form of W3C proposals only around 1998 - and because of the complex changes required to renderers, it took several years for reasonably robust implementations to become widespread.

There are three distinct ways to place CSS directives in HTML documents:

1. The use of an inline `STYLE="..."` parameter attached to HTML tags of any type; attributes specified this way apply to this and nested tags only (and largely defeat the purpose of the entire scheme when it comes to making it easy to alter the appearance of a document),
2. Introduction of a block of CSS code with `<STYLE>...</STYLE>` in any portion of the document. This block may change the default appearance of any tag, or define named rulesets that may be explicitly applied to specific tags with a `CLASS="..."` parameter,
3. Inclusion of a remote stylesheet with a `<LINK REL="stylesheet" HREF="...">`, with the same global effect as a `<STYLE>` block.

In the first two modes, the stylesheet generally inherits the character set of its host document, and is interpreted accordingly; in the last mode, character set might be derived from `Content-Type` headers, `@charset` directives, or auto-detected if all other options fail (reference).

Because CSS provides a powerful and standardized method to control the visual appearance of a block of HTML, many applications strive to let third parties control a subset of CSS syntax emitted in documents. Unfortunately, the task is fairly tricky; the most important security consequences of attacker-controlled stylesheets are:

- **The risk of JavaScript execution**. As a little-known feature, some CSS implementations permit JavaScript code to be embedded in stylesheets. There are at least three ways to achieve this goal: by using the `expression(...)` directive, which gives the ability to evaluate arbitrary JavaScript statements and use their value as a CSS parameter; by using the `url('javascript:...')` directive on properties that support it; or by invoking browser-specific features such as the `-moz-binding` mechanism of Firefox.

- **The ability to freely position text**. If user-controlled stylesheets are permitted on a page, various powerful CSS positioning directives may be invoked to move text outside the bounds of its current container, and mimick trusted UI elements or approximate them very accurately. Some examples of absolute positioning directives include `z-index`, `margin`, `padding`, `bottom`, `left`, `position`, `right`, `top`, or `text-indent`(many of them with sub-variants, such as `margin-left`).

- **The ability to reuse trusted classes**. If user-controlled `CLASS="..."` attributes are permitted in HTML syntax, the attacker may have luck "borrowing" a class used to render elements of the trusted UI and impersonate them.

Much like JavaScript, stylesheets are also tricky to sanitize, because they follow CDATA-style parsing: a literal sequence of `</STYLE>` ends the stylesheet regardless of its location within the CSS syntax as such. For example, the following stylesheet would be ended prematurely, and lead to JavaScript code being executed:

```
<STYLE>
body {
  background-image: url('http://example.com/foo.jpg?</STYLE><SCRIPT>alert(1)</SCRIPT>');
}
</STYLE>
```

Another interesting property specific to `<STYLE>` handling is that when two CDATA-like types overlap, no particular outcome is guaranteed; for example, the following may be interpreted as a script, or as an empty stylesheet, depending on the browser:

```
<STYLE>
<!-- </STYLE><SCRIPT>alert(1)</SCRIPT> -->
</STYLE>
```

Yet another characteristic that sets stylesheets apart from JavaScript is the fact that although CSS parser is very strict, a syntax error does not cause it to bail out completely. Instead, a recovery from the next top-level syntax element is attempted. This makes handling user-controlled strings even harder - for example, if a stray newline in user-supplied string is not properly escaped, it would actually permit the attacker to freely tinker with the stylesheet in most browsers:

```
<STYLE>
.example {
  content: '*** USER STRING START ***
  } .example { color: red; } .bar { *** USER STRING END ***';
}
</STYLE>

<SPAN CLASS="example">Hello world (in red)!</SPAN>
```

Additional examples along these lines are explored in more detail on this page. Several fundamental differences in style parsing between common browsers are outlined below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is JavaScript `expression(...)` supported?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Is script-targeted `url(...)` supported?	YES	NO	NO	NO	NO	NO	NO	NO	NO
Is script-executing `-moz-binding` supported?	NO	NO	NO	YES	NO	NO	NO	NO	NO
Does `</STYLE>` take precedence over comment block parsing?	NO	NO	NO	YES	YES	NO	NO	NO	NO
Characters permitted as CSS field-value separators (excluding `\t \r`	\x0B \x0C \	\x0B \x0C \	\x0B \x0C \	\x0B	\x0C	\x0C	\x0C \	\x0C \	\x0C \

\n \x20)	\xA0	\xA0	\xA0	\x0C \	\	\	\xA0	\xA0	

CSS character encoding

In many cases, as with JavaScript, there is a need for web applications to render certain user-supplied user-controlled strings within stylesheets in a safe manner. To handle various reserved characters, a method for escaping potentially troublesome values is required; confusingly, however, CSS format supports neither HTML entity encoding, nor any of the common methods of encoding characters seen in JavaScript.

Instead, a rather unusual and incompatible scheme consisting of `\` followed by non-prefixed, variable length one- to six-digit hexadecimal is employed; for example, `"test"` may be encoded as `"t\65st"` or `"t\000065st"` - but not as `t\est"`, `"t\x65st"`, `"t\u0065st"`, nor`"t&#x65;st"` (reference).

A very important and little-known oddity unique to CSS parsing is that escape sequences are also accepted outside strings, and confusingly, may substitute some syntax control characters in the stylesheet; so for example, `color: expression(alert(1))` and `color: expression \028 alert \028 1 \029 \029` have the same meaning. To add insult to injury, mixed syntax such as `color: expression ( alert \029 1 ) \029'` would not work.

With the exception of Internet Explorer, stray multi-line string literals are not supported; but a lone `\` at the end of a line may be used to seamlessly break long lines.

Other built-in document formats

HTML aside, modern browser renderers usually natively support an additional set of media formats that may be displayed as standalone documents. These can be generally divided into two groups:

- **Pure data formats.** These include plain text data or images (JPEG, GIF, BMP, etc), where the browser simply provides a basic rendering canvas and populates it with static data. In principle, no security consequences arise with these data types, as no malicious payloads may be embedded in the message - short of major and fairly rare implementation flaws. Common image formats aside, some browsers may also support other oddball or legacy media formats natively, such as the ability to play MID files specified by `<BGSOUND>` tags.

- **Rich data formats.** This category is primarily populated by non-HTML XML namespace parsers (SVG, RSS, Atom); beyond raw data, these document formats contain various rendering instructions, hints, or conditionals. Because of how XML works, each of the XML-based formats has two important security consequences:
  - Firstly, nested XML namespaces may be defined, and are usually not verified against MIME type intents, permitting HTML to be embedded for example inside `image/svg+xml`.
  - Secondly, these formats may actually come with provisions for non-standard embedded HTML or JavaScript payloads or scripts built in, permitting HTML injection even if the attacker has no direct control over XML document structure.

One example of a document that, even if served as `image/svg+xml`, would still execute scripts in many current browsers despite MIME type clearly stating a different intent, is as follows:

```
<?xml version="1.0"?>
<container>
  <svg xmlns="http://www.w3.org/2000/svg">
    [...]
  </svg>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <script>alert('Hello world!')</script>
  </html>
</container>
```

Furthermore, SVG natively permits embedded scripts and event handlers; in all browsers that support SVG, these scripts execute when the image is loaded as a top-level document - but are ignored when rendered through `<IMG>` tags.

Some of the non-HTML builtin document type behaviors are documented below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Supported bitmap formats (excluding JPG, GIF, PNG)	BMP ICO WMF	BMP ICO WMF	BMP ICO WMF	BMP ICO TGA*	BMP ICO TGA*	BMP TIF	BMP*	BMP ICO	BMP ICO
Is generic XML document support present?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is RSS feed support present?	NO	YES	YES	YES	YES	YES	YES	NO	NO
Is ATOM feed support present	NO	YES	YES	YES	YES	YES	YES	NO	NO

Does JavaScript execute within feeds?	(YES)	NO	NO	NO	NO	NO	NO	(YES)	(YES)
Are `javascript:` or `data:` URLs permitted in feeds?	n/a	NO	NO	NO	NO	YES	YES	n/a	n/a
Are CSS specifications permitted in feeds?	n/a	NO	YES	YES	YES	NO	YES	n/a	n/a
Is SVG image support present?	NO	NO	NO	YES	YES	YES	YES	YES	NO
May `image/svg+xml` document contain HTML `xmlns` payload?	(YES)	(YES)	(YES)	YES	YES	YES	YES	YES	(YES)

* Format support limited, inconsistent, or broken.

Trivia: curiously, Microsoft's XML-based Vector Markup Language (VML) is not natively supported by the renderer, and rather implemented as a plugin; whereas Scalable Vector Graphics (SVG) is implemented as a core renderer component in all browsers that support it.

Plugin-supported content

Unlike the lean and well-defined set of natively supported document formats, the landscape of browser plugins is extremely diverse, hairy, and evolving very quickly. Most of the common content-rendering plugins are invoked through the use of `<OBJECT>` or `<EMBED>` tags (or `<APPLET>`, for Java), but other types of integration may obviously take place.

Common document-embeddable plugins can be generally divided into several primary categories:

- **Web programming languages**. This includes technologies such as Adobe Flash, multi-vendor Java, or Microsoft Silverlight, together present at a vast majority of desktop computers. These languages generally permit extensive scripting, including access to the top-level document and other DOM information, or the ability to send network requests to at least some locations. The security models implemented by these plugins generally diverge from the models of the browser itself in sometimes counterintuitive or underspecified ways (discussed in more detail later on).

  This property makes such web development technologies a major attack surface by themselves if used legitimately. It also creates a security risk whenever a desire to user content arises, as extra steps must be taken to make sure the data could never be accidentally interpreted as a valid input to any of the popular plugins.

- **Drop-in integration for non-HTML document formats**. Some popular document editors or viewers, including Acrobat Reader and Microsoft Office, add the ability to embed their supported document formats as objects on HTML pages, or to view content, full-window, directly within the browser. The interesting property of this mechanism is that many such document formats either feature scripting capabilities, or offer interactive features (e.g., clickable links) that may result in data passed back to the browser in unusual or unexpected ways. For example, methods exist to navigate browser's window to `javascript:` URLs from PDF documents served inline - and this code will execute in the security context of the hosting domain.

- **Specialized HTML-integrated markup languages**. In addition to kludgy drop-in integration for document formats very different from HTML, specialized markup languages such as VRML, VML, or MathML, are also supported in some browsers through plugins, and meant to discretely supplement regular HTML documents; these may enable HTML smuggling vectors similar to those of renderer-handled XML formats (see previous section).

- **Rich multimedia formats**. A variety of plugins brings the ability to play video and audio clips directly within the browser, without opening a new media player window. Plugin-driven browser integration is offered by almost all contemporary media players, including Windows Media Player,QuickTime, RealPlayer, or VLC, again making it a widely available capability. Most of such formats bear no immediate security consequences for the hosting party per se, although in practice, this type of integration is plagued by implementation-specific bugs.

- **Specialized data manipulation widgets**. This includes features such as DHTML ActiveX editing gizmos (`2D360201-FFF5-11D1-8D03-00A0C959BC0A`). Some such plugins ship with Windows and are marked as safe despite receiving very little security scrutiny.

  Trivia: there is reportedly a sizeable and generally underresearched market of plugin-based crypto clients implemented as ActiveX controls in certain Asian countries.

One of the more important security properties of object-embedding tags is that like with many of their counterparts intended for embedding other non-HTML content, no particular attention is paid to server-supplied `Content-Type` and `Content-Disposition` headers, despite the fact that many embedded objects may in fact execute in a security context related to the domain that served them.

The precedence of inputs for deciding how to interpret the content in various browsers is as follows:

Input signal	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Tag type and `TYPE=` / `CLASSID=` values	#1	#1	#1	#1	#1	#1	#1	#1	n/a
`Content-Type` value if `TYPE=` is not recognized	ignored	ignored	ignored	ignored	ignored	ignored	ignored	ignored	n/a
`Content-Type`=value if `TYPE=` is missing	#2	#2	#2	#2	#2	#2	#2	#2	n/a
Content sniffing if `TYPE=` is not recognized	ignored	ignored	ignored	ignored	ignored	ignored	ignored	ignored	n/a
Content sniffing if `TYPE=` is missing	ignored	ignored	ignored	ignored	ignored	(#3)	ignored	(#3)	n/a

The approach to determining how to handle the data with no attention to the intent indicated by the hosting server is a problematic design concept, as it makes it impossible for servers to opt out from having any particular resource being treated as a plugin-interpreted document in response to actions of a malicious third-party site. In conjunction with lax syntax parsers within browser plugins, this resulted in a number of long-standing vulnerabilities, such as valid Java archive files (JARs) doubling as valid images; or, more recently, Flash applets doing the same.

Another interesting property worth noting here is that many plugins have their own HTTP stacks and caching systems, which makes them a good way to circumvent browser's privacy settings; Metasploit maintains an example site illustrating the problem.

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` dat a 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:** 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 are accepted in all 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 JavaScript-related problems:** the aforementioned `document.cookie` JavaScript 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	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 appears to be not given by Microsoft in MSDN documentation (reference).

- 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 inhttprequest.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	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 JavaScript 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.

- **JavaScript DOM access:** JavaScript 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 JavaScript. 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:

```
<script>
var location = "clobber";
</script>
<iframe src="http://www.example.com/frame_busting_code.html"></iframe>
```

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	NO	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`.

- **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[]` access third-party IFRAMEs?	YES	YES	NO	YES	YES	YES	NO	YES*	YES
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 targets?	NO	NO	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.location.*` properties be set on unrelated targets?	YES	YES	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.location.*` methods be called on unrelated targets?	YES	YES	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.document.write()` be called on unrelated targets?	NO	NO	NO	YES	NO	NO	NO	NO	NO
Can `TARGET=` links reposition unrelated targets?	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 JavaScript disabled, somewhat less efficient purely CSS-based enumeration is possible by referencing a unique server-side image via target-specific `:visited` *descriptors (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, "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.

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 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 - 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	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, 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); 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; another is the tendency for browsers to interpret completely non-HTTP responses as HTTP/0.9, covered here.

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.example.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 asses 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 reasonsnoted 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 ("leashing"), but permits sites to override this behavior by declaring a proper, user-friendly intent through compact P3P privacy policy headers (a mechanism discussed in more detail here and 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 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 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 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 (Firefox docs are an example). Following widespread complaints, current 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., image/* behavior changes), while other were resisted (e.g., 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	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 (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 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 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 (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: Internet Explorer list, Firefox list) based on very liberal and undocumented heuristics, historically including even parent character set inheritance (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.

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	NO	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 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. 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 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 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	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	NO	YES	YES	NO
Is `window.print()` limited or can be suppressed?	NO	NO	NO	NO	NO	NO	NO	NO	n/a

Is `window.confirm()` limited or can be suppressed?	NO	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 (reference), but in practice had the unfortunate effect of making it easy to spoof system prompts, mimick 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 RFC 2818, provides browsers and servers with the ability to establish encrypted TCP connections built on top of the 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 public key certificate, which would be then validated against a set of 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 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 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 KB931125 is installed, and browser's phishing filter functionality is enabled.

Trivia: 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.

Experimental and legacy security mechanisms

Through the years, browsers accrued a fair number of security mechanisms that had either fallen into disuse, or never caught on across more than a single vendor; as well as a number of ongoing proposals, enhancements, and extensions that are yet to prove their worth or become even vaguely standardized. This section provides a brief overview of several technologies that could fall into this class.

Fun fact: when it comes to newly proposed features, many of them essentially introduce new security boundaries and permission systems mostly orthogonal, yet intertwined, with same-origin controls. Although this appears to be a good idea at first sight, some researchers warn about the pitfalls of finer-grained origins as difficult to understand to users, and hard to fully examine for potential side effects and interactions with existing code.

HTTP authentication

HTTP authentication is an ancient mechanism most recently laid out in RFC 2617. It is a simple extension of HTTP:

- Any resource for which the server requires the user to provide valid credentials would initially return a `401 Unauthorized` HTTP error code, with a `WWW-Authenticate` header describing parameters such as the authentication realm, supported authentication modes, and other method-specific parameters.
- Upon receiving a `401` code, the client is expected to prompt the user for login and password, or obtain the data from in-memory cache or another credential store (where according to the RFC, it should be looked up by authentication realm name alone; although for security purposes, it should be bound to the host name as well, preferably also to port and protocol).
- User's credentials are then encoded and sent back in a new request with an additional `Authorization` header, which the server is expected to examine, and grant access or return an error message as appropriate.

Credentials are also cached for authentication with other subresources on the same site, and are sent out on future requests in an unsolicited manner (globally, or only for a particular path prefix).

Two key authentication schemes are supported by virtually all clients: `basic` and `digest`. The former simply sends user names and passwords in plain (`base64`) text - and hence the data is vulnerable to snooping, unless the process takes place over HTTPS. The latter uses a simple nonce-based challenge-response mechanism that prevents immediate password disclosure. Microsoft further extended the mechanism to include two proprietary `NTLM` and `Negotiate` schemes (reference) that integrate seamlessly with Microsoft Windows domain authentication.

As hinted in the section on URL syntax, URLs are permitted to have an optional `user[:password]@` segment immediately before the host name, to enable pre-authenticated bookmarks or shared links. In practice, the mechanism would be seldom used for legitimate purposes, but became immensely popular with phishers - who would often construct URLs such as `http://www.example-bank.com:something_something@www.evilsite.com/` in hopes of confusing the user. This led to this URL syntax being banned in Microsoft Internet Explorer, and often resulting in security prompts elsewhere.

Because of these limitations and the relative inflexibility of this scheme to begin with, HTTP authentication has been almost completely extinct on the Internet, and replaced with custom solutions built around HTTP cookies (it is still sometimes used for intranet applications or for simple access control for personal resources).

Amusingly, its ghost still haunts modern web applications: HTTP authentication prompts often come up in browsers when viewing trusted pages where a minor authentication-requiring sub-resource, such as `<IMG>`, is included from a rogue site - but these prompts usually do a poor job of clearly explaining who is asking for the credentials. This poses a phishing risk for services, such as blogs or discussion forums, that allow users to embed external content.

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is HTTP authentication supported?	YES	YES	YES	YES	YES	YES	YES	YES	NO
Does link-embedded authentication work?	NO	NO	NO	prompt	prompt	YES	prompt	YES	n/a
Is authentication data bound to realms?	NO	NO	NO	NO	NO	YES	NO	YES	n/a
Is authentication bound to host name?	YES	YES	YES	YES	YES	YES	YES	YES	n/a
Is authentication bound to protocol or port?	YES	YES	YES	YES	YES	YES	YES	YES	n/a
Are cached credentials sent out on all future requests?	YES	YES	YES	YES	YES	subdir only	subdir only	subdir only	n/a
Do password prompts come up on `<IMG>`?	YES	YES	YES	YES	YES	YES	YES	YES	n/a

Do password prompts come up on <EMBED> / <APPLET>?	NO	NO	NO	YES	YES	NO	YES	YES	n/a
Do password prompts come up on <SCRIPT> and stylesheets?	YES	YES	YES	YES	YES	YES	YES	YES	n/a
Do password prompts come up on <IFRAME>?	YES	YES	YES	YES	YES	YES	YES	YES	n/a

Name look-ahead and content prefetching

Several browsers are toying with the idea of prefetching certain information that, to their best knowledge, is likely to be needed in the immediate future. For example, Firefox offers an option to perform a background prefetch of certain links specified by page authors (reference); and Chrome is willing to carry out look-ahead DNS lookups on links on a page (reference). The idea here is that if the user indeed takes the action anticipated by browser developers or page owners, he or she would appreciate the reduced latency achieved through these predictive, background operations.

On the flip side, prefetching has two important caveats: one is that it is generally rather wasteful - as users are very unlikely to follow *all* prefetched links, and quite often, would not follow any of them; the other is that in certain applications, such look-aheads may reveal privacy-sensitive information to third parties.

One particularly interesting risk scenario is the context of a web mail interface: unless prefetching is properly disabled or limited, the sender of a mail containing a HTTP link might have the ability to detect that the recipient had read his message, even if the link itself is not followed.

Password managers

As a part of a broader form auto-completion mechanism, most browsers offer users the ability to save their passwords on client side, and auto-complete them whenever previously seen authentication forms appear again.

Unfortunately, as noted in the previous section, the only standard mechanism for identifying and scoping user passwords in HTTP traffic had largely fallen into disuse; the new form- and cookie-based alternatives are custom-built for every application, are share almost nothing in common with each other. As a result, most browsers face challenges trying heuristically detect when a legitimate authentication attempt takes place and succeeds, or trying to decide how to define the realm for stored data.

In particular, it might be easier than intuitively expected for certain types of user content hosted (or injected) under the same host name as a trusted login interface to spoof such forms and intercept auto-filled data; letting user-controlled forms and login interfaces mix within a single same-origin context appears to be not necessarily a good idea for time being.

Several of the relevant password manager behaviors are shown below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Password manager operation model	needs user name	needs user name	needs user name	auto-fills on load	auto-fills on load	auto-fills on load	needs UI action	auto-fills on load	auto-fills on load
Are stored passwords restricted to a full URL path?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Are stored https passwords restricted to SSL only?	YES	YES	YES	YES	YES	YES	YES	YES	NO

Robert Chapin offers some additional research into password manager scoping habits, exploring some non-security considerations as well.

Microsoft Internet Explorer zone model

All current versions of Internet Explorer utilize an interesting concept of security zones, not shared with any other browser on the market. The idea behind the approach is to compartmentalize resources into various categories, depending on the degree of trust given - and then control almost all security settings for these groups separately. The following zones are defined:

- **My computer:** a container for all local file:/// resources, with the exception of documents annotated with mark-of-the-web tags. The user has no control over what gets included into this zone, although an urlmon.dll API is provided to give certain protocol handlers the ability to define additional mappings (reference).

- **Local intranet:** a container for all sites determined by simple configurable heuristics to belong to the local network (non-FQDN host names, proxy server exception list, content accessed over SMB). The user may list additional host names to include in this zone.

- **Trusted sites:** an empty container for trusted pages. This group enjoys certain elevated privileges, particularly to run ActiveX controls, install desktop elements, or programatically access the clipboard. The user is expected to populate the zone with trusted sites that require these permissions to operate.

- **Restricted sites:** an empty container for untrusted pages. This group has many of the rudimentary permissions taken away, such as the ability to initiate file downloads or use `<META HTTP-EQUIV="Refresh" ...>`. The user is expected to populate the zone with sites he would rather approach with caution (presumably ahead of the first planned visit).

- **Internet:** a default container for all sites on the Internet not included in any of the remaining categories.

This design makes it possible to, for example, fine-tune the permissions of `file:///` resources without impacting Internet sites, or to forbid navigation from "Internet" to "Local intranet" - and from this perspective, appears to offer a major security benefit. On the flip side:

- Quite a few important security settings are excluded from the zone model, somewhat diminishing its value; for example, cookie permissions or SSL behavior is controlled elsewhere.

- The number of settings currently offered in this model is remarkably high, and many of them appear to be too finely grained, or have vague or confusing descriptions with security impact not clearly explained (*"Script controls marked as safe for scripting"*, *"Navigate sub-frames across different domains"*), increasing the risk of hard-to-spot configuration errors.

- The model fails to account for the impact of cross-site scripting flaws in trusted sites. Since users add pages such as Windows Update, their banks, and other legacy sites that may not always work properly in the "Internet" zone to "Trusted sites", giving them a lot of unnecessary permissions as a side effect, the impact of cross-site scripting flaws on these pages may result in the attacker suddenly gaining the ability to carry out dangerous actions normally not available to Internet content.

- The complexity of the model and the permissive settings of some zones resulted in a large number of security problems that could easily be avoided by employing a simpler approach instead.

Microsoft Internet Explorer frame restrictions

Another interesting, little-known security feature unique to Microsoft Internet Explorer is the concept of SECURITY=RESTRICTED frames. The idea behind the mechanism is that some services may have a legitimate use for limiting the ability of data displayed within `<IFRAME>` containers to disrupt the top-level document or abuse same-origin policies - and so, with the `SECURITY=RESTRICTED` attribute, the content may be placed in the "Restricted sites" zone, and - in theory - deprived of the ability to run disruptive scripts or access cookies.

No support outside of Microsoft Internet Explorer makes this feature useless as a security defense for most intents and purposes; but the mechanism needs to be considered for its potential negative security impact, such as the ability to prevent frame busting code from operating properly and making UI redress attacks a bit easier. No support for HTTP cookies within a restricted container would normally limit the impact, but flaws in the design are known.

Due to its minimal use, the mechanism likely received very little security scrutiny otherwise.

HTML5 sandboxed frames

A better-developed descendant of the `SECURITY=RESTRICTED` is the current HTML5 proposal for sandboxed frames. The design appears to be considerably more robust and offers a better granularity for restricting the behavior of framed sites: embedding domains will be able to disallow scripting altogether; allow JavaScript but make all same-origin policy checks fail; or allow scripting, but prevent navigating the top-level document to prevent framebusting.

Of all these features, the ability to place content in a same-origin policy sandbox is the most tricky one. Perhaps most importantly, the attacker could simply open the normally framed document directly (this is prevented by using `text/html-sandboxed` as a MIME type; but content sniffing logic will render this trick unsafe in certain browsers). The other significant problem is that mechanisms such as local storage and workers (see next chapter), form autocomplete, and so forth, need to be specifically accounted forth and denied access to - something that will likely prove challenging in the long run.

HTML5 storage, cache, and worker experiments

Firefox 2 embraced the idea of globalStorage, a somewhat haphazard mechanism that permitted data to be stored persistently for offline use in a local database shared across all sites (and until Firefox 2, with no strong security controls on scoping). The concept of `globalStorage` originated with early HTML5 drafts, although there, got eventually ditched in favor of a more tightly controlled localStorage and SQL database schemes (yet to be implemented in Firefox, though the the latter option is already supported in a handful of WebKit browsers and Microsoft Internet Explorer 8).

Further along these lines, Firefox 3 introduced, and Firefox 3.1 revised in a largely incompatible manner, the concept of cache manifests that permit sites to opt for having certain resources persistently stored in a site-specific cache, and then retrieved from there instead of being looked up on the net. The feature is again borrowed from the ever-shifting HTML5 specification (and now also available in WebKit browsers).

Web workers are another HTML5 design along these lines, starting to ship with WebKit browsers. The idea is to permit asynchronous, background JavaScript execution in a separate process; dedicated workers would be tied to their opener, and terminated when the page closes; shared workers are not bound to their origin, and free to

use `postMessage` API to interact with third-party domains; and persistent workers may be allowed to launch and execute across browser sessions with no additional dependencies of any sort. There are still some kinks in the design documents at this point, however.

At this point, it is not entirely clear how useful and widely used these storage and script execution mechanisms would be, or how they would evolve. Local storage proposals with no cross-domain access facilities are largely a non-security-relevant mechanism, spare for privacy considerations and the risk of implementation flaws; shared storage and shared workers are a bit more tricky, for obvious reasons; and cache manifests are troubling in that they may make cache poisoning attacks a bit easier and more predictable.

Microsoft Internet Explorer XSS filtering

An experimental reflected HTML injection filter is available in Microsoft Internet Explorer 8. The filter attempts to disable any Javascript on the page that seems to be copied from query parameters. The complexity of escaping rules, jiffy handling of certain character sets, and quirky rules for parsing HTML documents, all make it likely that the mechanism would never achieve a 100% coverage - a property that Microsoft pragmatically acknowledges in their design proposals. As such, the mechanism would likely serve to complement, rather than replace, proper server-side development practices.

An important downside of the design is that it aims to selectively disable script snippets that appear in URL query parameters, instead of displaying an interstitial or other security prompt, and permitting the page to display in full, or not display at all. This behavior may potentially enable attackers to carefully take out frame-busting code or other security-critical components of targeted applications, while leaving the rest of the functionality intact.

The filter also features a same-origin exception that inhibits filtering when site-supplied links are followed. In many complex web applications, this may be used to bypass the mechanism altogether.

Script restriction frameworks

Several experimental browser-side or client-side designs aim to provide control over same-origin permissions for scripts, or over when scripts may execute on pages to begin with. Although such features would not necessarily prevent all the potential negative effects of rendering attacker-controlled HTML in trusted domains, a well-executed solution could indeed avert most of the high-impact vulnerabilities.

There are two primary classes of solutions proposed:

- **Comprehensive script security frameworks:** these approaches strive to offer the ability for site owner to approve or reject arbitrary actions executed by scripts with a great deal of flexibility. Examples include Microsoft Mutation-Event Transforms for browser-side checks (not clear if this is being pursued in any browser), or Google Caja and Microsoft Web Sandbox for a solution that requires no browser-side tweaks.

- **Selective script disable / enable functions:** these comparatively simpler solutions attempt to give web developers the tools to disable scripting in certain critical areas of the document, such as around user input or a block of intentionally non-sanitized HTML. One example such a proposal are Trevor Jim's Browser-Enforced Embedded Policies, or the toStaticHTML() API in Microsoft Internet Explorer 8.

Secure JSON parsing

A traditional method for modern web applications to parse same-origin, serialized text JavaScript responses returned by servers in response to `XMLHttpRequest` calls is to pass them to `eval()` - which turns the string into a native object. Unfortunately, calls to `eval()` have a number of side effects - most notably, any code smuggled inside the string would execute in the recipient script security context, putting an additional burden on the server or the client to sanitize the code - and leading to many security bugs.

Several implementations of secure JSON parsers were proposed earlier, such as the (unsafe!) implementation suggested RFC4627, but they generally combine varying levels of insecurity with very significant performance penalties.

In response to this, several browsers rolled out JSON.parse(), a native non-executing JSON parser that could be used as a drop-in replacement for `eval()`. The interface does not address the much greater risk of loading cross-domain JavaScript information via `<SCRIPT SRC=...>` or similar approaches, and the cross-browser support is at this point very limited, shipping in Chrome and Firefox 3.5.

Origin headers

Adam Barth, Collin Jackson, and other researchers propose the introduction of reliable Origin headers as an answer to the risk of cross-site request forgery - giving sites the ability to reliably and easily decide whether a particular request permitted across domains comes from a trusted party or not.

In theory, an existing `Referer` HTTP header could be used for the same purpose, but many users install tweaks to suppress it for privacy reasons; the header may also be dropped on certain page transitions, and was not generally designed to be reliable. `Origin` proposals attempt to limit the risk of the header being disabled by limiting the amount of data being sent across domains (host name, but no full URL; though arguably, it may still easily disclose undesirable information); and limiting the types of requests on which the header is provided (although this may undermine some of the security benefits associated with the solution).

Mozilla content security policies

Brandon Sterne of Mozilla proposes content security policies (specification), a mechanism that would permit site owners to define a list of valid target domains for `<IMG>`, `<EMBED>`, `<APPLET>`, `<SCRIPT>`, `<IFRAME>`, and similar resources appearing in their content; and to disallow inline code from being used directly on a page.

The security benefit of these features is limited primarily to cross-site scripting prevention and webmaster policy enforcement; the proposal originally also incorporated the ability to define a list of valid origins for incoming requests as a method to mitigate cross-site request forgery attacks - but this part of the design got dropped in favor of more flexible and simpler `Origin` headers.

Open browser engineering issues

Other than the general design of HTTP, HTML, and related mechanisms discussed previously, a handful of browser engineering decisions tends to contribute to a disproportional number of day-to-day security issues. Understanding these properties is sometimes important for properly assessing the likelihood and maximum impact of security breaches, and hence determining the safety of user data. Some of the pivotal, open-ended issues include:

- **Relatively unsafe core programming languages:** C++ is used for a majority of code in Internet Explorer, Firefox, Safari, Opera, and Chrome; C is used in certain high-performance or low-level areas, such as image manipulation libraries. The choice of C and C++ means that browsers are regularly plagued by memory management and integer overflow problems, despite considerable ongoing audit efforts.

- **No security compartmentalization:** once control of the process is seized due to common implementation flaws, most browsers provide essentially unconstrained access to the user context they are running in. This means that browser bugs - historically, very common - easily lead to total system integrity loss.

  There are three exceptions: Chrome uses a security sandbox to contain the renderer (which includes WebKit and V8, hence constituting one of the largest and most complicated bodies of code in the project). Access to system functions and browser-managed data is heavily constrained, and individual tabs are generally run in separate processes, with some exceptions for new tabs spawned in response to document navigation. Android, on the other hand, runs the browser in a dedicated user account, and then restricts the ability for this context to execute any undesirable actions in the system. Likewise, Internet Explorer 7, when running on Windows Vista, is running with reduced privileges as a whole (reference)..

- **Web technologies are used in browser chrome:** JavaScript, HTML, and XML are all used to a varying degree to implement some browser internals and various diagnostic and error pages in most browsers. This choice contributes to an elevated risk of HTML injection flaws that permit web content to gain elevated chrome privileges, which - depending on the browser - may carry the permission to read or write files, access arbitrary sites on the Internet, or alter browser settings. The problem is particularly pronounced for Firefox, which implements much of its user interface in this manner.

- **Inconsistent and overly complex security UIs:** a vast majority of browsers employ highly inconsistent UI elements and security messaging, including several styles of modal prompts, interstitials, icons, color codes, and messages that pop up either on the bottom or on the top of the document window. Usability studies consistently show that at least some of these features are easily mis-identified, misunderstood, or trivial to spoof (this is particularly the case for interstitials and notification bars that are not anchored in browser UI). Although a gradual improvement may be observed in certain aspects, further coordinated work in this area seems to be necessary; timing attack issues are a particularly interesting problem to deal with.

- **Ad hoc plugin security models:** every plugin enforces its own variant of the same-origin policy and HTTP stack security checks, often with subtle variations from the browser implementation, and with a relatively poor track record. There is no compelling reason why these checks should not be managed centrally in browser code, other than the absence of well-documented APIs. This situation leads to frequent plugin security model issues, and make web application development more challenging than needs to be.

- **Inconsistent and haphazard data storage practices:** browsers use a mix of random storage methods to keep temporary files, downloads, configuration data, and sensitive records such as passwords, browsing history, saved cookies, or cache entries. These methods include system registry, database container files, drop-off directories, text-based configs (CSV, INI, tab-delimited, XML), and proprietary binary files. The data may be stored in user home directories, system-wide temporary directories, or global program installation folders. Controlling the permissions on all these resources and manipulating them securely is relatively difficult, contributing to many problems, particularly in multi-user systems, or when multiple browsers are used by the same user.

- **General susceptibility to denial-of-service attacks:** as discussed in the section on JavaScript execution restrictions, browsers are generally vulnerable to resource exhaustion or UI blocking attacks that may crash or render the browser - or even the underlying operating system - unresponsive or difficult to interact with. In some cases, this may be even abused to extort certain UI actions from less experienced users. Even with JavaScript out of the picture, tasks such as parsing XML or HTML documents, or rendering certain types of images or videos, are often sufficiently computationally expensive to facilitate such attacks.

- **Incompatibility with untrusted networks:** browsers will persistently cache resources retrieved over untrusted networks, such as public wifi - and will grant such content full access to previously cached pages or credentials. This problem makes it virtually impossible to use public wireless hotspots safely.