READ-ONLY: This project has been archived. For more information see this post.

Part1
Browser Security Handbook, part 1

Updated Mar 6, 2011 by lcam...@gmail.com

Browser Security Handbook, part 1

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

Table of Contents

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_
```

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 3492 outlines 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 4395 mandated 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 and XHTML (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.

A fascinating quirk exists in the Internet Explorer HTML parser. While all other browsers accept quoted string parameter values only if the quote appear in front of the parameter, MSIE looks for an *equals-quote* substring anywhere in the middle:

```
<img src=test.jpg?value=">Yes, we are still inside a tag!">
```

Other behaviors unique to Internet Explorer include the recognition of conditional HTML syntax, such as:

```
<!--[if IE 6]>
  Markup that will be parsed only for Internet Explorer 6
<![endif]-->
```

...and the recognition of %-type, comment-like HTML tags.

Some of the other 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	NO	NO	NO	NO
Recursive recovery with nested tags (both FOO and BAR interpreted)?	(NO)	(NO)	(NO)	YES	YES	(NO)	(NO)	(NO)	(NO)
Parser resets out on invalid tag names (`<FOO=" <BAR...`)?	NO	NO	NO	YES	YES	NO	NO	NO	NO
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
!- and ?-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

For a good overview of these and many other HTML parsing quirks, you should probably have a look at "Web Application Obfuscation", a book by Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Heyes, and David Lindsay.

NOTE 1: 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.

NOTE 2: the behavior of HTML parsers, WebKit in particular, is changing rapidly due to some of the HTML5 work.

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 `&lt;` for <, `&gt;` for >, `&rarr;` for →, etc,
- Decimal entities, &#<nn>;, with a number corresponding to the desired Unicode character value - for example `&#60;` for <, `&#8594;` for →,
- Hexadecimal entities, &#x<nn>;, likewise - for example `&#x3c;` for <, `&#x2192;` 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&#09;: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	∞	∞	∞	∞	∞	∞
Maximum length of an incorrectly terminated decimal entity	7	7	7	∞	∞	∞	∞	∞	∞
Maximum length of a correctly terminated hex entity	6	6	6	∞	∞	∞	∞	∞	∞
Maximum length of an incorrectly terminated hex entity	0	0	0	∞	∞	∞	∞	∞	∞

Does entity value wraps around 2^32?	NO	NO	NO	NO	NO	YES	NO	YES	YES
Characters permitted in entity names (excluding `A-Z a-z 0-9`)	none	none	none	`-.`	`-.`	none	none	none	none

In WebKit, entries past a certain length used to get parsed, but incorrectly; for example, `&#000000065;` would become a sequence of three characters, `\x06 5 ;`. This has been fixed recently.

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 the current 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 \n \x20)	\x0B \x0C \ \xA0	\x0B \x0C \ \xA0	\x0B \x0C \ \xA0	\x0B \x0C \	\x0C \	\x0C \	\x0C \ \xA0	\x0C \ \xA0	\x0C \

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	NO	NO	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 guaranteed to 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.

NOTE: Unfortunately, as of today, the embedding party is also not in complete control of how a plugin-handled document will be displayed. See this article for more.

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.

(Continue to browser security features...)