HTTP
Internet-Draft
Obsoletes: [6265] (if approved)
Intended status: Standards Track
Expires: 22 May 2026

A. van Kesteren, Ed.
Apple
J. Hofmann, Ed.
Google

18 November 2025

Cookies: HTTP State Management Mechanism
draft-ietf-httpbis-layered-cookies-01

Abstract

This document defines the HTTP Cookie and Set-Cookie header fields. These header fields can be used by HTTP servers to store state (called cookies) at HTTP user agents, letting the servers maintain a stateful session over the mostly stateless HTTP protocol. Although cookies have many historical infelicities that degrade their security and privacy, the Cookie and Set-Cookie header fields are widely used on the Internet. This document obsoletes RFC 6265 and 6265bis.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-httpbis-layered-cookies/.

Discussion of this document takes place on the HTTP Working Group mailing list (mailto:ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/. Working Group information can be found at https://httpwg.org/.

Source for this draft and an issue tracker can be found at https://github.com/httpwg/http-extensions/labels/cookies.

Status of This Memo

This Internet-Draft will expire on 22 May 2026.

Copyright Notice

Table of Contents

1. Introduction

This document defines the HTTP Cookie and Set-Cookie header fields. Using the Set-Cookie header field, an HTTP server can pass name/value pairs and associated metadata (called cookies) to a user agent. When the user agent makes subsequent requests to the server, the user agent uses the metadata and other information to determine whether to return the name/value pairs in the Cookie header field.

Although simple on their surface, cookies have a number of complexities. For example, the server can scope the maximum amount of time during which the user agent should return the cookie, the servers to which the user agent should return the cookie, and whether the cookie can be accessed through a non-HTTP API, such as JavaScript in a web browser.

For historical reasons, cookies contain a number of security and privacy infelicities. For example, a server can indicate that a given cookie is intended for "secure" connections, but the cookie's Secure attribute does not provide integrity in the presence of an active network attacker. Similarly, cookies for a given host are shared across all the ports on that host, even though the usual "same-origin policy" used by web browsers isolates content retrieved via different ports.

This document specifies the syntax and semantics of these header fields. Where some existing software differs from the requirements in significant ways, the document contains a note explaining the difference.

This document obsoletes [RFC6265] and 6265bis.

1.1. Examples

Using the Set-Cookie header field, a server can send the user agent a short string in an HTTP response that the user agent will return in future HTTP requests that are within the scope of the cookie. For example, the server can send the user agent a "session identifier" named SID with the value 31d4d96e407aad42. The user agent then returns the session identifier in subsequent requests.

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

The server can alter the default scope of the cookie using the Path and Domain attributes. For example, the server can instruct the user agent to return the cookie to every path and every subdomain of site.example.

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=site.example

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

As shown in the next example, the server can store multiple cookies at the user agent. For example, the server can store a session identifier as well as the user's preferred language by returning two Set-Cookie header fields. Notice that the server uses the Secure and HttpOnly attributes to provide additional security protections for the more sensitive session identifier (see Section 4.1.2).

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
Set-Cookie: lang=en-US; Path=/; Domain=site.example

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42; lang=en-US
```

Notice that the Cookie header field above contains two cookies, one named SID and one named lang.

Cookie names are case-sensitive, meaning that if a server sends the user agent two Set-Cookie header fields that differ only in their name's case the user agent will store and return both of those cookies in subsequent requests.

```
== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42
Set-Cookie: sid=31d4d96e407aad42

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42; sid=31d4d96e407aad42
```

If the server wishes the user agent to persist the cookie over multiple "sessions" (e.g., user agent restarts), the server can specify an expiration date in the Expires attribute. Note that the user agent might delete the cookie before the expiration date if the user agent's cookie store exceeds its quota or if the user manually deletes the server's cookie.

```
== Server -> User Agent ==

Set-Cookie: lang=en-US; Expires=Wed, 09 Jun 2021 10:18:14 GMT

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42; lang=en-US
```

Finally, to remove a cookie, the server returns a Set-Cookie header field with an expiration date in the past. The server will be successful in removing the cookie only if the Path and the Domain attribute in the Set-Cookie header field match the values used when the cookie was created.

```
== Server -> User Agent ==

Set-Cookie: lang=; Expires=Sun, 06 Nov 1994 08:49:37 GMT

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
```

2. Conventions

2.1. Terminology

This specification depends on Infra. [INFRA]

Some terms used in this specification are defined in the following standards and specifications:

* HTTP [RFC9110]

* URL [URL]

A *non-HTTP API* is a non-HTTP mechanisms used to set and retrieve cookies, such as a web browser API that exposes cookies to JavaScript.

2.2. ABNF

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

The following core rules are included by reference, as defined in [RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTLs (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), NUL (null octet), OCTET (any 8-bit sequence of data except NUL), SP (space), HTAB (horizontal tab), CHAR (any ASCII byte), VCHAR (any visible ASCII byte), and WSP (whitespace).

The OWS (optional whitespace) and BWS (bad whitespace) rules are defined in Section 5.6.3 of [RFC9110].

3. Which Requirements to Implement

The upcoming two sections, Section 4 and Section 5, discuss the set of requirements for two distinct types of implementations. This section is meant to help guide implementers in determining which set of requirements best fits their goals. Choosing the wrong set of requirements could result in a lack of compatibility with other cookie implementations.

It’s important to note that being compatible means different things depending on the implementer’s goals. These differences have built up over time due to both intentional and unintentional specification changes, specification interpretations, and historical implementation differences.

This section roughly divides implementers of this specification into two types, producers and consumers. These are not official terms and are only used here to help readers develop an intuitive understanding of the use cases.

3.1. Cookie Producing Implementations

An implementer should choose Section 4 whenever cookies are created and will be sent to a user agent, such as a web browser. These implementations are frequently referred to as servers by the specification but that term includes anything which primarily produces cookies. Some potential examples:

* Server applications hosting a website or API

* Programming languages or software frameworks that support cookies

* Integrated third-party web applications, such as a business management suite

All these benefit from not only supporting as many user agents as possible but also supporting other servers. This is useful if a cookie is produced by a software framework and is later sent back to a server application which needs to read it. Section 4 advises best practices that help maximize this sense of compatibility.

3.2. Cookie Consuming Implementations

An implementer should choose Section 5 whenever cookies are primarily received from another source. These implementations are referred to as user agents. Some examples:

* Web browsers

* Tools that support stateful HTTP

* Programming languages or software frameworks that support cookies

Because user agents don’t know which servers a user will access, and whether or not that server is following best practices, users agents are advised to implement a more lenient set of requirements and to accept some things that servers are warned against producing. Section 5 advises best practices that help maximize this sense of compatibility.

3.3. Programming Languages & Software Frameworks

A programming language or software framework with support for cookies could reasonably be used to create an application that acts as a cookie producer, cookie consumer, or both. Because a developer may want to maximize their compatibility as either a producer or consumer, these languages or frameworks should strongly consider supporting both sets of requirements, Section 4 and Section 5, behind

a compatibility mode toggle. This toggle should default to Section 4's requirements.

Doing so will reduce the chances that a developer's application can inadvertently create cookies that cannot be read by other servers.

4. Server Requirements

This section describes the conforming syntax and semantics of the HTTP Cookie and Set-Cookie header fields.

4.1. Set-Cookie

The Set-Cookie HTTP response header field is used to send cookies from the server to the user agent.

Origin servers MAY send a Set-Cookie response header field with any response. An origin server can include multiple Set-Cookie header fields in a single response. The presence of a Cookie or a Set-Cookie header field does not preclude HTTP caches from storing and reusing a response.

Origin servers and intermediaries MUST NOT combine multiple Set-Cookie header fields into a single header field. The usual mechanism for combining HTTP headers fields (i.e., as defined in Section 5.3 of [RFC9110]) might change the semantics of the Set-Cookie header field because the %x2C (",") character is used by Set-Cookie in a way that conflicts with such combining.

For example,

Set-Cookie: a=b;path=/c,d=e

is ambiguous. It could be intended as two cookies, a=b and d=e, or a single cookie with a path of /c,d=e.

4.1.1. Syntax

Informally, the Set-Cookie response header field contains a cookie, which begins with a name-value-pair, followed by zero or more attribute-value pairs. Servers MUST send Set-Cookie header fields that conform to the following grammar:

```
set-cookie        = set-cookie-string
set-cookie-string = BWS cookie-pair *( BWS ";" OWS cookie-av )
cookie-pair       = cookie-name BWS "=" BWS cookie-value
cookie-name       = token
cookie-value      = *cookie-octet / ( DQUOTE *cookie-octet DQUOTE )
cookie-octet      = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E
                      ; US-ASCII characters excluding CTLs,
                      ; whitespace, DQUOTE, comma, semicolon,
                      ; and backslash
token             = <token, defined in [HTTP], Section 5.6.2>

cookie-av         = expires-av / max-age-av / domain-av /
                    path-av / secure-av / httponly-av /
                    samesite-av / extension-av
expires-av        = "Expires" BWS "=" BWS sane-cookie-date
sane-cookie-date  =
    <IMF-fixdate, defined in [HTTP], Section 5.6.7>
max-age-av        = "Max-Age" BWS "=" BWS non-zero-digit *DIGIT
non-zero-digit    = %x31-39
                      ; digits 1 through 9
domain-av         = "Domain" BWS "=" BWS domain-value
domain-value      = <subdomain>
                      ; see details below
path-av           = "Path" BWS "=" BWS path-value
path-value        = *av-octet
secure-av         = "Secure"
httponly-av       = "HttpOnly"
samesite-av       = "SameSite" BWS "=" BWS samesite-value
samesite-value    = "Strict" / "Lax" / "None"
extension-av      = *av-octet
av-octet          = %x20-3A / %x3C-7E
                      ; any CHAR except CTLs or ";"
```

Note that some of the grammatical terms above reference documents that use different grammatical notations than this document (which uses ABNF from [RFC5234]).

Per the grammar above, servers MUST NOT produce nameless cookies (i.e., an empty cookie-name) as such cookies may be unpredictably serialized by user agents when sent back to the server.

The semantics of the cookie-value are not defined by this document.

To maximize compatibility with user agents, servers that wish to store arbitrary data in a cookie-value SHOULD encode that data, for example, using Base64 [RFC4648].

Per the grammar above, the cookie-value MAY be wrapped in DQUOTE characters. Note that in this case, the initial and trailing DQUOTE characters are not stripped. They are part of the cookie-value, and will be included in Cookie header fields sent to the server.

The domain-value is a subdomain as defined by [RFC1034], Section 3.5, and as enhanced by [RFC1123], Section 2.1. Thus, domain-value is a byte sequence of ASCII bytes.

The portions of the set-cookie-string produced by the cookie-av term are known as attributes. To maximize compatibility with user agents, servers SHOULD NOT produce two attributes with the same name in the same set-cookie-string.

NOTE: The name of an attribute-value pair is not case-sensitive. So while they are presented here in CamelCase, such as HttpOnly or SameSite, any case is accepted. E.g., httponly, Httponly, hTTPoNLY, etc.

Servers MUST NOT include more than one Set-Cookie header field in the same response with the same cookie-name. (See Section 5.5.1 for how user agents handle this case.)

If a server sends multiple responses containing Set-Cookie header fields concurrently to the user agent (e.g., when communicating with the user agent over multiple sockets), these responses create a "race condition" that can lead to unpredictable behavior.

NOTE: Some existing user agents differ in their interpretation of two-digit years. To avoid compatibility issues, servers SHOULD use the rfc1123-date format, which requires a four-digit year.

NOTE: Some user agents store and process dates in cookies as 32-bit UNIX time_t values. Implementation bugs in the libraries supporting time_t processing on some systems might cause such user agents to process dates after the year 2038 incorrectly.

4.1.2. Semantics (Non-Normative)

This section describes simplified semantics of the Set-Cookie header field. These semantics are detailed enough to be useful for understanding the most common uses of cookies by servers. The full semantics are described in Section 5.

When the user agent receives a Set-Cookie header field, the user agent stores the cookie together with its attributes. Subsequently, when the user agent makes an HTTP request, the user agent includes the applicable, non-expired cookies in the Cookie header field.

If the user agent receives a new cookie with the same cookie-name, domain-value, and path-value as a cookie that it has already stored, the existing cookie is evicted and replaced with the new cookie. Notice that servers can delete cookies by sending the user agent a new cookie with an Expires attribute with a value in the past.

Unless the cookie's attributes indicate otherwise, the cookie is returned only to the origin server (and not, for example, to any subdomains), and it expires at the end of the current session (as defined by the user agent). User agents ignore unrecognized cookie attributes (but not the entire cookie).

4.1.2.1. The Expires Attribute

The Expires attribute indicates the maximum lifetime of the cookie, represented as the date and time at which the cookie expires. The user agent is not required to retain the cookie until the specified date has passed. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

4.1.2.2. The Max-Age Attribute

The Max-Age attribute indicates the maximum lifetime of the cookie, represented as the number of seconds until the cookie expires. The user agent is not required to retain the cookie for the specified duration. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

If a cookie has both the Max-Age and the Expires attribute, the Max-Age attribute has precedence and controls the expiration date of the cookie. If a cookie has neither the Max-Age nor the Expires attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent).

4.1.2.3. The Domain Attribute

The Domain attribute specifies those hosts to which the cookie will be sent.

If the server includes the Domain attribute, the value applies to both the specified domain and any subdomains. For example, if the value of the Domain attribute is "site.example", the user agent will include the cookie in the Cookie header field when making HTTP requests to site.example, www.site.example, and www.corp.site.example. Note that a leading %x2E ("."), if present, is ignored even though that character is not permitted.

If the server omits the Domain attribute, the user agent will return the cookie only to the origin server and not to any subdomains.

WARNING: Some existing user agents treat an absent Domain attribute as if the Domain attribute were present and contained the current host name. For example, if site.example returns a Set-Cookie header field without a Domain attribute, these user agents will erroneously send the cookie to www.site.example and www.corp.site.example as well.

The user agent will reject cookies unless the Domain attribute specifies a scope for the cookie that would include the origin server. For example, the user agent will accept a cookie with a Domain attribute of "site.example" or of "foo.site.example" from foo.site.example, but the user agent will not accept a cookie with a Domain attribute of "bar.site.example" or of "baz.foo.site.example".

4.1.2.4. The Path Attribute

The scope of each cookie is limited to a set of paths, controlled by the Path attribute. If the server omits the Path attribute, the user agent will use the "directory" of the request-uri's path component as the default value. (See Section 5.3.3 for more details.)

The user agent will include the cookie in an HTTP request only if the path portion of the request-uri matches (or is a subdirectory of) the cookie's Path attribute, where the %x2F ("/") character is interpreted as a directory separator.

Although seemingly useful for isolating cookies between different paths within a given host, the Path attribute cannot be relied upon for security (see Section 8).

4.1.2.5. The Secure Attribute

The Secure attribute limits the scope of the cookie to "secure" channels (where "secure" is outside the scope of this document). E.g., when a cookie has the Secure attribute, the user agent will include the cookie in an HTTP request only if the request is transmitted over a secure channel (typically HTTP over Transport Layer Security (TLS) [TLS13] [RFC9110]).

4.1.2.6. The HttpOnly Attribute

The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via non-HTTP APIs.

4.1.2.7. The SameSite Attribute

The SameSite attribute limits the scope of the cookie upon creation and delivery with respect to whether the cookie is considered to be "same-site" within a larger context (where "same-site" is outside the scope of this document). The SameSite attribute is particularly relevant for web browsers and web applications accessible through them.

The SameSite attribute supports Strict, Lax, and None as values.

4.1.3. Cookie Name Prefixes

Section 8.5 and Section 8.6 of this document spell out some of the drawbacks of cookies' historical implementation. In particular, it is impossible for a server to have confidence that a given cookie was set with a particular set of attributes. In order to provide such confidence in a backwards-compatible way, two common sets of requirements can be inferred from the first few characters of the cookie's name.

To maximize compatibility with user agents, servers SHOULD use prefixes as described below.

4.1.3.1. The "__Secure-" Prefix

If a cookie's name begins with a case-sensitive match for the string __Secure-, then the cookie will have been set with a Secure attribute.

For example, the following Set-Cookie header field would be rejected by a conformant user agent, as it does not have a Secure attribute.

Set-Cookie: __Secure-SID=12345; Domain=site.example

Whereas the following Set-Cookie header field would be accepted if set from a secure origin (e.g. https://site.example/), and rejected otherwise:

Set-Cookie: __Secure-SID=12345; Domain=site.example; Secure

4.1.3.2. The "__Host-" Prefix

If a cookie's name begins with a case-sensitive match for the string __Host-, then the cookie will have been set with a Secure attribute, a Path attribute with a value of /, and no Domain attribute.

This combination yields a cookie that hews as closely as a cookie can to treating the origin as a security boundary. The lack of a Domain attribute ensures that cookie's host-only is true, locking the cookie to a particular host, rather than allowing it to span subdomains. Setting the Path to / means that the cookie is effective for the entire host, and won't be overridden for specific paths. The Secure attribute ensures that the cookie is unaltered by non-secure origins, and won't span protocols.

Ports are the only piece of the same-origin policy that __Host-cookies continue to ignore.

For example, the following cookies would always be rejected:

```
Set-Cookie: __Host-SID=12345
Set-Cookie: __Host-SID=12345; Secure
Set-Cookie: __Host-SID=12345; Domain=site.example
Set-Cookie: __Host-SID=12345; Domain=site.example; Path=/
Set-Cookie: __Host-SID=12345; Secure; Domain=site.example; Path=/
```

While the following would be accepted if set from a secure origin (e.g. https://site.example/), and rejected otherwise:

```
Set-Cookie: __Host-SID=12345; Secure; Path=/
```

4.1.3.3. The "__Http-" Prefix

If a cookie's name begins with a case-sensitive match for the string __Http-, then the cookie will have been set with a Secure attribute, and an HttpOnly attribute.

This helps developers and server operators to know that the cookie was set using a Set-Cookie header, and is limited in scope to HTTP requests.

4.1.3.4. The "__Host-Http-" prefix

If a cookie's name begins with a case-sensitive match for the string __Host-Http-, then the cookie will have been set with a Secure attribute, an HttpOnly attribute, a Path attribute with a value of /, and no Domain attribute.

This helps developers and server operators to know that the cookie was set using a Set-Cookie header, and is limited in scope to HTTP requests.

This combination yields a cookie that hews as closely as a cookie can to treating the origin as a security boundary, while at the same time ensuring developers and server operators know that its scope is limited to HTTP requests. The lack of a Domain attribute ensures that cookie's host-only is true, locking the cookie to a particular host, rather than allowing it to span subdomains. Setting the Path to / means that the cookie is effective for the entire host, and won't be overridden for specific paths. The Secure attribute ensures that the cookie is unaltered by non-secure origins, and won't span protocols. The HttpOnly attribute ensures that the cookie is not exposed by the user agent to non-HTTP APIs.

4.2. Cookie

4.2.1. Syntax

The user agent sends stored cookies to the origin server in the Cookie header field. If the server conforms to the requirements in Section 4.1 (and the user agent conforms to the requirements in Section 5), the user agent will send a Cookie header field that conforms to the following grammar:

```
cookie        = cookie-string
cookie-string = cookie-pair *( ";" SP cookie-pair )
```

While Section 5.4 of [RFC9110] does not define a length limit for header fields it is likely that the web server's implementation does impose a limit; many popular implementations have default limits of 8 kibibytes. Servers SHOULD avoid setting a large number of large cookies such that the final cookie-string would exceed their header field limit. Not doing so could result in requests to the server failing.

Servers MUST be tolerant of multiple Cookie headers. For example, an HTTP/2 [RFC9113] or HTTP/3 [HTTP] client or intermediary might split a Cookie header to improve compression. Servers are free to determine what form this tolerance takes. For example, the server could process each Cookie header individually or the server could concatenate all the Cookie headers into one and then process that final, single, header. The server should be mindful of any header field limits when deciding which approach to take.

Note: Since intermediaries can modify Cookie headers they should also be mindful of common server header field limits in order to avoid sending servers headers that they cannot process. For example, concatenating multiple cookie headers into a single header might exceed a server's size limit.

4.2.2. Semantics

Each cookie-pair represents a cookie stored by the user agent. The cookie-pair contains the cookie-name and cookie-value the user agent received in the Set-Cookie header field.

Notice that the cookie attributes are not returned. In particular, the server cannot determine from the Cookie field alone when a cookie will expire, for which hosts the cookie is valid, for which paths the cookie is valid, or whether the cookie was set with the Secure or HttpOnly attributes.

The semantics of individual cookies in the Cookie header field are not defined by this document. Servers are expected to imbue these cookies with application-specific semantics.

Although cookies are serialized linearly in the Cookie header field, servers SHOULD NOT rely upon the serialization order. In particular, if the Cookie header field contains two cookies with the same name (e.g., that were set with different Path or Domain attributes), servers SHOULD NOT rely upon the order in which these cookies appear in the header field.

5. User Agent Requirements

This section specifies the processing models associated with the Cookie and Set-Cookie header fields in sufficient detail that a user agent can interoperate with existing servers (even those that do not conform to the well-behaved profile described in Section 4).

A user agent could enforce more restrictions than those specified herein (e.g., restrictions specified by its cookie policy, described in Section 7.2). However, such additional restrictions may reduce the likelihood that a user agent will be able to interoperate with existing servers.

5.1. Cookie Concepts

To facilitate the algorithms that follow, a number of pre-requisite concepts need to be introduced.

5.1.1. Cookie Store And Limits

A user agent has an associated *cookie store*, which is a list of cookies. It is initially 束 損.

A user agent has an associated *total cookies-per-host limit*, which is an integer. It SHOULD be 50 or more.

A user agent has an associated *total cookies limit*, which is an integer. It SHOULD be 3000 or more.

A user agent has an associated *cookie age limit*, which is a number of days. It SHOULD be 400 days or less (see Section 7.2).

5.1.2. Cookie Struct

A *cookie* is a struct that represents a piece of state to be transmitted between a client and a server.

A cookie's *name* is a byte sequence. It always needs to be set.

A cookie's *value* is a byte sequence. It always needs to be set.

A cookie's *secure* is a boolean. It is initially false.

A cookie's *host* is a domain, IP address, null, or failure. It is initially null. Note: Once a cookie is in the user agent's cookie store its host is a domain or IP address.

A cookie's *host-only* is a boolean. It is initially false.

A cookie's *path* is a URL path.

A cookie's *has-path attribute* is a boolean. It is initially false.

A cookie's *same-site* is "strict", "lax", "unset", or "none". It is initially "unset".

A cookie's *http-only* is a boolean. It is initially false.

A cookie's *creation-time* is a time. It is initially the current time.

A cookie's *expiry-time* is null or a time. It is initially null. Note: A prior version of this specification referred to null with a distinct "persistent-flag" field being false.

A cookie's *last-access-time* is a time. It is initially the current time.

5.1.2.1. Cookie Struct Miscellaneous

A cookie is *expired* if its expiry-time is non-null and its expiry-time is in the past.

A cookie _cookie_ is *Host-prefix compatible* if:

1. _cookie_’s secure is true;

2. _cookie_’s host-only is true;

3. _cookie_’s has-path attribute is true; and

4. _cookie_’s path’s size is 1 and _cookie_’s path[0] is the empty string,

A cookie _cookie_ is *Http-prefix compatible* if:

1. _cookie_’s secure is true; and

2. _cookie_’s http-only is false,

5.2. Cookie Store Eviction

5.2.1. Remove Expired Cookies

1. Let _expiredCookies_ be a list of references to all expired cookies in the user agent’s cookie store.

2. For each _cookie_ of _expiredCookies_:

    1. Remove _cookie_ from the user agent’s cookie store.

3. Return _expiredCookies_.

5.2.2. Remove Excess Cookies for a Host

To *Remove Excess Cookies for a Host* given a host _host_:

1. Let _insecureCookies_ be a list of references to all cookies in the user agent’s cookie store whose host is host-equal to _host_ and whose secure is false.

2. Sort _insecureCookies_ by earliest last-access-time first.

3. Let _secureCookies_ be a list of references to all cookies in the user agent’s cookie store whose host is host-equal to _host_ and whose secure is true.

4. Sort _secureCookies_ by earliest last-access-time first.

5. Let _excessHostCookies_ be an empty list of cookies.

6. While _insecureCookies_’s size + _secureCookies_’s size is greater than the user agent’s total cookies-per-host limit:

1. If _insecureCookies_ is not empty:
    1. Let _cookie_ be the first item of _insecureCookies_.
    2. Remove _cookie_ from _insecureCookies_.
    3. Remove _cookie_ from the user agent’s cookie store.
    4. Append _cookie_ to _excessHostCookies_.
2. Otherwise:
    1. Let _cookie_ be the first item of _secureCookies_.
    2. Remove _cookie_ from _secureCookies_.
    3. Remove _cookie_ from the user agent’s cookie store.
    4. Append _cookie_ to _excessHostCookies_.

7. Return _excessHostCookies_.

5.2.3. Remove Global Excess Cookies

To *Remove Global Excess Cookies*:

1. Let _allCookies_ be the result of sorting the user agent’s cookie store by earliest last-access-time first.
2. Let _excessGlobalCookies_ be an empty list of cookies.
3. While _allCookies_’s size is greater than the user agent’s total cookies limit:
    1. Let _cookie_ be the first item of _allCookies_.
    2. Remove _cookie_ from _allCookies_.
    3. Remove _cookie_ from the user agent’s cookie store.
    4. Append _cookie_ to _excessGlobalCookies_.
4. Return _excessGlobalCookies_.

5.3. Subcomponent Algorithms

This section defines some algorithms used by user agents to process specific subcomponents of the Cookie and Set-Cookie header fields.

5.3.1. Parse a Date

To *Parse a Date* given a byte sequence _input_, run these steps. They return a date and time or failure.

Note that the various boolean flags defined as a part of the algorithm (i.e., found-time, found-day-of-month, found-month, found-year) are initially "not set".

1. Using the grammar below, divide the cookie-date into date-tokens.

```
cookie-date     = *delimiter date-token-list *delimiter
date-token-list = date-token *( 1*delimiter date-token )
date-token      = 1*non-delimiter

delimiter       = %x09 / %x20-2F / %x3B-40 / %x5B-60 / %x7B-7E
non-delimiter   = %x00-08 / %x0A-1F / DIGIT / ":" / ALPHA
                  / %x7F-FF
non-digit       = %x00-2F / %x3A-FF

day-of-month    = 1*2DIGIT [ non-digit *OCTET ]
month           = ( "jan" / "feb" / "mar" / "apr" /
                    "may" / "jun" / "jul" / "aug" /
                    "sep" / "oct" / "nov" / "dec" ) *OCTET
year            = 2*4DIGIT [ non-digit *OCTET ]
time            = hms-time [ non-digit *OCTET ]
hms-time        = time-field ":" time-field ":" time-field
time-field      = 1*2DIGIT
```

2. Process each date-token sequentially in the order the date-tokens appear in the cookie-date:

    1. If the found-time flag is not set and the token matches the time production, set the found-time flag and set the hour-value, minute-value, and second-value to the numbers denoted by the digits in the date-token, respectively. Skip the remaining sub-steps and continue to the next date-token.

    2. If the found-day-of-month flag is not set and the date-token matches the day-of-month production, set the found-day-of-month flag and set the day-of-month-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

    3. If the found-month flag is not set and the date-token matches the month production, set the found-month flag and set the month-value to the month denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

4. If the found-year flag is not set and the date-token matches the year production, set the found-year flag and set the year-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

3. If the year-value is greater than or equal to 70 and less than or equal to 99, increment the year-value by 1900.

4. If the year-value is greater than or equal to 0 and less than or equal to 69, increment the year-value by 2000.

    1. NOTE: Some existing user agents interpret two-digit years differently.

5. If one of the following is true:

    * at least one of the found-day-of-month, found-month, found-year, or found-time flags is not set,

    * the day-of-month-value is less than 1 or greater than 31,

    * the year-value is less than 1601,

    * the hour-value is greater than 23,

    * the minute-value is greater than 59, or

    * the second-value is greater than 59,

    then return failure.

    (Note that leap seconds cannot be represented in this syntax.)

6. Let the parsed-cookie-date be the date whose day-of-month, month, year, hour, minute, and second (in UTC) are the day-of-month-value, the month-value, the year-value, the hour-value, the minute-value, and the second-value, respectively. If no such date exists, abort these steps and fail to parse the cookie-date.

7. Return the parsed-cookie-date as the result of this algorithm.

5.3.2. Domain Matching

A host _host_ *Domain-Matches* a string _domainAttributeValue_ if at least one of the following is true:

* _host_ equals _domainAttributeValue_, or

* if all of the following are true:

  - _host_ is a domain, and

  - _host_ ends with the concatenation of U+002E (.) and _domainAttributeValue_.

5.3.3. Cookie Default Path

To determine the *Cookie Default Path*, given a URL path _path_, run these steps. They return a URL path.

1. Assert: _path_ is a non-empty list.

2. If _path_'s size is greater than 1, then remove _path_'s last item.

3. Otherwise, set _path_[0] to the empty string.

4. Return _path_.

5.3.4. Path Matching

To determine if a URL path _requestPath_ *Path-Matches* a URL path _cookiePath_, run these steps. They return a boolean.

1. Let _serializedRequestPath_ be the result of URL path serializing _requestPath_.

2. Let _serializedCookiePath_ be the result of URL path serializing _cookiePath_.

3. If _serializedCookiePath_ is _serializedRequestPath_, then return true.

4. If _serializedRequestPath_ starts with _serializedCookiePath_ and _serializedCookiePath_ ends with a U+002F (/), then return true.

5. Return whether the concatenation of _serializedRequestPath_ followed by U+002F (/) starts with _serializedCookiePath_.

5.4. Main Algorithms

5.4.1. Parse and Store a Cookie

To *Parse and Store a Cookie* given a byte sequence _input_, boolean _isSecure_, domain or IP address _host_, URL path _path_, boolean _httpOnlyAllowed_, boolean _allowNonHostOnlyCookieForPublicSuffix_, and boolean _sameSiteStrictOrLaxAllowed_:

1. Let _cookie_ be the result of running Parse a Cookie with _input_, _isSecure_, _host_, and _path_.

2. If _cookie_ is failure, then return.

3. Return the result of Store a Cookie given _cookie_, _isSecure_, _host_, _path_, _httpOnlyAllowed_, _allowNonHostOnlyCookieForPublicSuffix_, and _sameSiteStrictOrLaxAllowed_.

5.4.2. Parse a Cookie

To *Parse a Cookie* given a byte sequence _input_, boolean _isSecure_, host _host_, URL path _path_, run these steps. They return a new cookie or failure:

1. If _input_ contains a byte in the range 0x00 to 0x08, inclusive, the range 0x0A to 0x1F inclusive, or 0x7F (CTL bytes excluding HTAB), then return failure.

2. Let _nameValueInput_ be null.

3. Let _attributesInput_ be the empty byte sequence.

4. If _input_ contains 0x3B (;), then set _nameValueInput_ to the bytes up to, but not including, the first 0x3B (;), and _attributesInput_ to the remainder of _input_ (including the 0x3B (;) in question).

5. Otherwise, set _nameValueInput_ to _input_.

6. Assert: _nameValueInput_ is a byte sequence.

7. Let _name_ be null.

8. Let _value_ be null.

9. If _nameValueInput_ does not contain a 0x3D (=) character, then set _name_ to the empty byte sequence, and _value_ to _nameValueInput_.

10. Otherwise, set _name_ to the bytes up to, but not including, the first 0x3D (=), and set _value_ to the bytes after the first 0x3D (=) (possibly being the empty byte sequence).

11. Remove any leading or trailing WSP bytes from _name_ and _value_.

12. If _name_'s length + _value_'s length is 0 or is greater than 4096, then return failure.

13. Let _cookie_ be a new cookie whose name is _name_ and value is _value_.

14. Set _cookie_'s path to the result of running Cookie Default Path with _path_.

    Note: A Path attribute can override this.

15. While _attributesInput_ is not an empty byte sequence:

    1. Let _maxAgeSeen_ be false.

    2. Let _char_ be the result of consuming the first byte of _attributesInput_.

    3. Assert: _char_ is 0x3B (;).

    4. Let _attributeNameValueInput_ be null.

    5. If _attributesInput_ contains 0x3B (;), then set _attributeNameValueInput_ to the result of consuming the bytes of _attributesInput_ up to, but not including, the first 0x3B (;).

    6. Otherwise, set _attributeNameValueInput_ to the result of consuming the remainder of _attributesInput_.

    7. Let _attributeName_ be null.

    8. Let _attributeValue_ be the empty string.

    9. If _attributeNameValueInput_ contains a 0x3D (=), then set _attributeName_ to the bytes up to, but not including, the first 0x3D (=) of _attributeNameValueInput_, and _attributeValue_ to the bytes after the first 0x3D (=) of _attributeNameValueInput_.

10. Otherwise, set _attributeName_ to _attributeNameValueInput_.

11. Remove any leading or trailing WSP bytes from _attributeName_ and _attributeValue_.

12. If _attributeValue_’s length is greater than 1024, then continue.

13. If _attributeName_ is a byte-case-insensitive match for Expires:

    1. If _maxAgeSeen_ is true, then continue.

    2. Let _expiryTime_ be the result of running Parse a Date given _attributeValue_.

    3. If _attributeValue_ is failure, then continue.

    4. If _expiryTime_ is greater than the current time and date + the user agent’s cookie age limit, then set _expiryTime_ to the user agent’s cookie age limit.

    5. If _expiryTime_ is earlier than the earliest date the user agent can represent, the user agent MAY replace _expiryTime_ with the earliest representable date.

    6. Set _cookie_’s expiry-time to _expiryTime_.

14. If _attributeName_ is a byte-case-insensitive match for Max-Age:

    1. If _attributeValue_ is empty, continue.

    2. If the first byte of _attributeValue_ is neither a DIGIT, nor 0x2D (-) followed by a DIGIT, then continue.

    3. If the remainder of _attributeValue_ contains a non-DIGIT, then continue.

    4. Let _deltaSeconds_ be _attributeValue_, converted to a base 10 integer.

    5. Set _deltaSeconds_ to the smaller of _deltaSeconds_ and the user agent’s cookie age limit, in seconds.

6. If _deltaSeconds_ is less than or equal to 0, let _expiryTime_ be the earliest representable date and time. Otherwise, let _expiryTime_ be the current date and time + _deltaSeconds_ seconds.

7. Set _cookie_'s expiry-time to _expiryTime_.

8. Set _maxAgeSeen_ to true.

15. If _attributeName_ is a byte-case-insensitive match for Domain:

    1. Let _host_ be failure.

    2. If _attributeValue_ contains only ASCII bytes:

        1. Let _hostInput_ be _attributeValue_, ASCII decoded.

        2. If _hostInput_ starts with U+002E (.), then set _hostInput_ to _hostInput_ without its leading U+002E (.).

        3. Set _host_ to the result of host parsing _hostInput_.

    3. Set _cookie_'s host to _host_.

16. If _attributeName_ is a byte-case-insensitive match for Path:

    1. If _attributeValue_ is not empty and if the first byte of _attributeValue_ is 0x2F (/), then:

        1. Set _cookie_'s path to _attributeValue_ split on 0x2F (/).

        2. Set _cookie_'s has-path attribute to true.

17. If _attributeName_ is a byte-case-insensitive match for Secure:

    1. Set _cookie_'s secure to true.

18. If _attributeName_ is a byte-case-insensitive match for HttpOnly:

    1. Set _cookie_'s http-only to true.

19. If _attributeName_ is a byte-case-insensitive match for SameSite:

    1. If _attributeValue_ is a byte-case-insensitive match for None, then set _cookie_'s same-site to "none".

    2. If _attributeValue_ is a byte-case-insensitive match for Strict, then set _cookie_'s same-site to "strict".

    3. If _attributeValue_ is a byte-case-insensitive match for Lax, then set _cookie_'s same-site to "lax".

16. Return _cookie_.

Note: Attributes with an unrecognized _attributeName_ are ignored.

Note: This intentionally overrides earlier cookie attributes so that generally the last specified cookie attribute "wins".

5.4.3. Store a Cookie

To *Store a Cookie* given a cookie _cookie_, boolean _isSecure_, domain or IP address _host_, boolean _httpOnlyAllowed_, boolean _allowNonHostOnlyCookieForPublicSuffix_, and boolean _sameSiteStrictOrLaxAllowed_:

1. Assert: _cookie_'s name's length + _cookie_'s value's length is not 0 or greater than 4096.

2. Assert: _cookie_'s name does not contain a byte in the range 0x00 to 0x08, inclusive, in the range 0x0A to 0x1F, inclusive, or 0x7F (CTL characters excluding HTAB).

3. If _cookie_'s host is failure, then return null.

4. Set _cookie_'s creation-time and last-access-time to the current date and time.

5. If _allowNonHostOnlyCookieForPublicSuffix_ is false and _cookie_'s host is a public suffix:

    1. If _cookie_'s host is host-equal to _host_, then set _cookie_'s host to null.

    2. Otherwise, return null.

Note: This step prevents attacker.example from disrupting the integrity of site.example by setting a cookie with a Domain attribute of example. In the event the end user navigates directly to example, a cookie can still be set and will forcibly have its host-only set to true.

6. If _cookie_'s host is null:

    1. Set _cookie_'s host-only to true.

    2. Set _cookie_'s host to _host_.

7. Otherwise:

    1. If _host_ does not Domain-Match _cookie_'s host, then return null.

    2. Set _cookie_'s host-only to false.

8. Assert: _cookie_'s host is a domain or IP address.

9. If _httpOnlyAllowed_ is false and _cookie_'s http-only is true, then return null.

10. If _isSecure_ is false:

    1. If _cookie_'s secure is true, then return null.

    2. If the user agent's cookie store contains at least one cookie _existingCookie_ that meets all of the following criteria:

        1. _existingCookie_'s name is _cookie_'s name;

        2. _existingCookie_'s secure is true;

        3. _existingCookie_'s host Domain-Matches _cookie_'s host, or vice-versa; and

        4. _cookie_'s path Path-Matches _existingCookie_'s path,

        then return null.

Note: The path comparison is not symmetric, ensuring only that a newly-created, non-secure cookie does not overlay an existing secure cookie, providing some mitigation against cookie-fixing attacks. That is, given an existing secure cookie named 'a' with a path of '/login', a non-secure cookie named 'a' could be set for a path of '/' or '/foo', but not for a path of '/login' or '/login/en'.

11. If _cookie_'s same-site is not "none" and _sameSiteStrictOrLaxAllowed_ is false, then return null.

12. If _cookie_'s same-site is "none" and _cookie_'s secure is false, then return null.

13. If _cookie_'s name, byte-lowercased, starts with __secure- and _cookie_'s secure is false, then return null.

    Note: The check here and those below are with a byte-lowercased value in order to protect servers that process these values in a case-insensitive manner.

14. If _cookie_'s name, byte-lowercased, starts with __host- and _cookie_ is not Host-prefix compatible, then return null.

15. If _cookie_'s name, byte-lowercased, starts with __http- and _cookie_ is not Http-prefix compatible, then return null.

16. If _cookie_'s name, byte-lowercased, starts with __host-http- and _cookie_ is not both Host-prefix compatible and Http-prefix compatible, then return null.

17. If _cookie_'s name is the empty byte sequence and one of the following is true:

    * _cookie_'s value, byte-lowercased, starts with __secure-,

    * _cookie_'s value, byte-lowercased, starts with __host-,

    * _cookie_'s value, byte-lowercased, starts with __http-, or

    * _cookie_'s value, byte-lowercased, starts with __host-http-,

    then return null.

18. If the user agent's cookie store contains a cookie _oldCookie_ whose name is _cookie_'s name, host is host-equal to _cookie_'s host, host-only is _cookie_'s host-only, and path is path-equal to _cookie_'s path:

1. If _httpOnlyAllowed_ is false and _oldCookie_'s http-only is true, then return null.

2. If _cookie_'s secure is equal to _oldCookie_'s secure, _cookie_'s same-site is equal to _oldCookie_'s same-site, and _cookie_'s expiry-time is equal to _oldCookie_'s expiry-time, then return null.

3. Set _cookie_'s creation-time to _oldCookie_'s creation-time.

4. Remove _oldCookie_ from the user agent's cookie store.

Note: This algorithm maintains the invariant that there is at most one such cookie.

19. Insert _cookie_ into the user agent's cookie store.

20. Return _cookie_.

5.4.4. Garbage Collect Cookies

To *Garbage Collect Cookies* given a host _host_:

1. Let _expiredCookies_ be the result of running Remove Expired Cookies.

2. Let _excessHostCookies_ be the result of running Remove Excess Cookies for Host given _host_.

3. Let _excessGlobalCookies_ be the result of running Remove Global Excess Cookies.

4. Let _removedCookies_ be 束 損 .

5. For each _cookieList_ of 束 _expiredCookies_, _excessHostCookies_, _excessGlobalCookies_ 損, do the following:

    1. Extend _removedCookies_ with _cookieList_.

6. Return _removedCookies_

5.4.5. Retrieve Cookies

To *Retrieve Cookies* given a boolean _isSecure_, host _host_, URL path _path_, boolean _httpOnlyAllowed_, and string _sameSite_:

1. Assert: _sameSite_ is "strict-or-less", "lax-or-less", "unset-or-less", or "none".

2. Let _cookies_ be all cookies from the user agent’s cookie store that meet these conditions:

   * One of the following is true:

     - cookie’s host-only is true and _host_ is host-equal to cookie’s host, or

     - cookie’s host-only is false and _host_ Domain-Matches cookie’s host.

     It’s possible that the public suffix list changed since a cookie was created. If this change results in a cookie’s host becoming a public suffix and the cookie’s host-only is false, then that cookie SHOULD NOT be returned.

     XXX: We should probably move this requirement out-of-bound as this invalidation should happen as part of updating the public suffixes.

   * _path_ Path-Matches cookie’s path.

   * One of the following is true:

     - cookie’s secure is true and _isSecure_ is true, or

     - cookie’s secure is false.

   * One of the following is true:

     - cookie’s http-only is true and _httpOnlyAllowed_ is true, or

     - cookie’s http-only is false.

   * One of the following is true:

     - cookie’s same-site is "strict" and _sameSite_ is "strict-or-less";

     - cookie’s same-site is "lax" and _sameSite_ is one of "strict-or-less" or "lax-or-less";

     - cookie’s same-site is "unset" and _sameSite_ is one of "strict-or-less", "lax-or-less", or "unset-or-less"; or

     - cookie’s same-site is "none".

3. Sort _cookies_ in the following order:

    * Cookies whose path's size is greater are listed before cookies whose path's size is smaller.

    * Among cookies whose path's size is equal, cookies whose creation-time is earlier are listed before cookies whose creation-time is later.

4. Set the last-access-time of each cookie of _cookies_ to the current date and time and reflect these changes in the user agent's cookie store.

5. Return _cookies_.

5.4.6. Serialize Cookies

To *Serialize Cookies* given a list of cookies _cookies_:

1. Let _output_ be an empty byte sequence.

2. For each _cookie_ of _cookies_:

    1. If _output_ is not the empty byte sequence, then append 0x3B (;) followed by 0x20 (SP) to _output_.

    2. If _cookie_'s name is not the empty byte sequence, then append _cookie_'s name followed by 0x3D (=) to _output_.

    3. Append _cookie_'s value to _output_.

3. Return _output_.

5.5. Requirements Specific to Non-Browser User Agents

5.5.1. The Set-Cookie Header Field

When a user agent receives a Set-Cookie header field in an HTTP response, the user agent MAY ignore the Set-Cookie header field in its entirety as per its cookie policy (see Section 7.2).

User agents MAY ignore Set-Cookie header fields contained in responses with 100-level status codes.

Set-Cookie header fields contained in responses with non-100-level status codes (including those in responses with 400- and 500-level status codes) SHOULD be processed as follows:

1. Let _isSecure_ be a boolean indicating whether request's URL's scheme is deemed secure, in an implementation-defined manner.

2. Let _host_ be request's host.

3. Let _path_ be request's URL's path.

4. Let _httpOnlyAllowed_ be true.

5. Let _allowNonHostOnlyCookieForPublicSuffix_ be a boolean whose value is implementation-defined.

6. Let _sameSiteStrictOrLaxAllowed_ be a boolean whose value is implementation-defined.

7. Let _cookie_ be the result of running Parse and Store a Cookie given the header field value, _isSecure_, _host_, _path_, _httpOnlyAllowed_, _allowNonHostOnlyCookieForPublicSuffix_, and _sameSiteStrictOrLaxAllowed_.

8. If _cookie_ is null, then return.

9. Run Garbage Collect Cookies given _cookie_'s host.

5.5.2. The Cookie Header Field

The user agent includes stored cookies in the Cookie request header field.

When the user agent generates an HTTP request, the user agent MUST NOT attach more than one Cookie header field.

A user agent MAY omit the Cookie header field in its entirety.

If the user agent does attach a Cookie header field to an HTTP request, the user agent MUST compute its value as follows:

1. Let _isSecure_ be a boolean indicating whether request's URL's scheme is deemed secure, in an implementation-defined manner.

2. Let _host_ be request's host.

3. Let _path_ be request's URL's path.

4. Let _httpOnlyAllowed_ be true.

5. Let _sameSite_ be a string whose value is implementation-defined, but has to be one of "strict-or-less", "lax-or-less", "unset-or-less", or "none".

6. Let _cookies_ be the result of running Retrieve Cookies given _isSecure_, _host_, _path_, _httpOnlyAllowed_, and _sameSite_.

7. Return the result of running Serialize Cookies given _cookies_.

Note: Previous versions of this specification required that only one Cookie header field be sent in requests. This is no longer a requirement. While this specification requires that a single cookie-string be produced, some user agents may split that string across multiple Cookie header fields. For examples, see Section 8.2.3 of [RFC9113] and Section 4.2.1 of [HTTP].

5.5.3. Cookie Store Eviction for Non-Browser User Agents

The user agent SHOULD evict all expired cookies from its cookie store if, at any time, an expired cookie exists in the cookie store, by calling Remove Expired Cookies.

When "the current session is over" (as defined by the user agent), the user agent MUST remove from the cookie store all cookies whose expiry-time is null.

5.6. Requirements Specific to Browser User Agents

While browsers are expected to generally follow the same model as non-browser user agents, they have additional complexity due to the document model (and the ability to nest documents) that is considered out-of-scope for this specification.

Specifications for such a user agent are expected to build upon the following algorithms and invoke them appropriately to process Cookie and Set-Cookie header fields, as well as manipulating the user agent’s cookie store through non-HTTP APIs:

* Parse and Store a Cookie

* Store a Cookie

* Remove Expired Cookies

* Garbage Collect Cookies

* Retrieve Cookies

* Serialize Cookies

This provides the flexibility browsers need to detail their requirements in considerable detail.

6. Implementation Considerations

6.1. Limits

Servers SHOULD use as few and as small cookies as possible to avoid reaching these implementation limits, minimize network bandwidth due to the Cookie header field being included in every request, and to avoid reaching server header field limits (See Section 4.2.1).

Servers SHOULD gracefully degrade if the user agent fails to return one or more cookies in the Cookie header field because the user agent might evict any cookie at any time.

6.2. Application Programming Interfaces

One reason the Cookie and Set-Cookie header fields use such esoteric syntax is that many platforms (both in servers and user agents) provide a string-based application programming interface (API) to cookies, requiring application-layer programmers to generate and parse the syntax used by the Cookie and Set-Cookie header fields, which many programmers have done incorrectly, resulting in interoperability problems.

Instead of providing string-based APIs to cookies, platforms would be well-served by providing more semantic APIs. It is beyond the scope of this document to recommend specific API designs, but there are clear benefits to accepting an abstract "Date" object instead of a serialized date string.

7. Privacy Considerations

Cookies’ primary privacy risk is their ability to correlate user activity. This can happen on a single site, but is most problematic when activity is tracked across different, seemingly unconnected Web sites to build a user profile.

Over time, this capability (warned against explicitly in [RFC2109] and all of its successors) has become widely used for varied reasons including:

* authenticating users across sites,

* assembling information on users,

* protecting against fraud and other forms of undesirable traffic,

* targeting advertisements at specific users or at users with specified attributes,

* measuring how often ads are shown to users, and

* recognizing when an ad resulted in a change in user behavior.

While not every use of cookies is necessarily problematic for privacy, their potential for abuse has become a widespread concern in the Internet community and broader society. In response to these concerns, user agents have actively constrained cookie functionality in various ways (as allowed and encouraged by previous specifications), while avoiding disruption to features they judge desirable for the health of the Web.

It is too early to declare consensus on which specific mechanism(s) should be used to mitigate cookies' privacy impact; user agents' ongoing changes to how they are handled are best characterised as experiments that can provide input into that eventual consensus.

Instead, this document describes limited, general mitigations against the privacy risks associated with cookies that enjoy wide deployment at the time of writing. It is expected that implementations will continue to experiment and impose stricter, more well-defined limitations on cookies over time. Future versions of this document might codify those mechanisms based upon deployment experience. If functions that currently rely on cookies can be supported by separate, targeted mechanisms, they might be documented in separate specifications and stricter limitations on cookies might become feasible.

Note that cookies are not the only mechanism that can be used to track users across sites, so while these mitigations are necessary to improve Web privacy, they are not sufficient on their own.

7.1. Third-Party Cookies

A "third-party" or cross-site cookie is one that is associated with embedded content (such as scripts, images, stylesheets, frames) that is obtained from a different server than the one that hosts the primary resource (usually, the Web page that the user is viewing). Third-party cookies are often used to correlate users' activity on different sites.

Because of their inherent privacy issues, most user agents now limit third-party cookies in a variety of ways. Some completely block third-party cookies by refusing to process third-party Set-Cookie header fields and refusing to send third-party Cookie header fields. Some partition cookies based upon the first-party context, so that different cookies are sent depending on the site being browsed. Some block cookies based upon user agent cookie policy and/or user controls.

While this document does not endorse or require a specific approach, it is RECOMMENDED that user agents adopt a policy for third-party cookies that is as restrictive as compatibility constraints permit. Consequently, resources cannot rely upon third-party cookies being treated consistently by user agents for the foreseeable future.

7.2. Cookie Policy

User agents MAY enforce a cookie policy consisting of restrictions on how cookies may be used or ignored (see Section 5.5.1).

A cookie policy may govern which domains or parties, as in first and third parties (see Section 7.1), for which the user agent will allow cookie access. The policy can also define limits on cookie size, cookie expiry (see Section 4.1.2.1 and Section 4.1.2.2), and the number of cookies per domain or in total.

The goal of a restrictive cookie policy is often to improve security or privacy. User agents often allow users to change the cookie policy (see Section 7.3).

7.3. User Controls

User agents SHOULD provide users with a mechanism for managing the cookies stored in the cookie store. For example, a user agent might let users delete all cookies received during a specified time period or all the cookies related to a particular domain. In addition, many user agents include a user interface element that lets users examine the cookies stored in their cookie store.

User agents SHOULD provide users with a mechanism for disabling cookies. When cookies are disabled, the user agent MUST NOT include a Cookie header field in outbound HTTP requests and the user agent MUST NOT process Set-Cookie header fields in inbound HTTP responses.

User agents MAY offer a way to change the cookie policy (see Section 7.2).

User agents MAY provide users the option of preventing persistent storage of cookies across sessions. When configured thusly, user agents MUST treat all received cookies as if their expiry-time is null.

7.4. Expiration Dates

Although servers can set the expiration date for cookies to the distant future, most user agents do not actually retain cookies for multiple decades. Rather than choosing gratuitously long expiration periods, servers SHOULD promote user privacy by selecting reasonable cookie expiration periods based on the purpose of the cookie. For example, a typical session identifier might reasonably be set to expire in two weeks.

8. Security Considerations

8.1. Overview

Cookies have a number of security pitfalls. This section overviews a few of the more salient issues.

In particular, cookies encourage developers to rely on ambient authority for authentication, often becoming vulnerable to attacks such as cross-site request forgery [CSRF]. Also, when storing session identifiers in cookies, developers often create session fixation vulnerabilities.

Transport-layer encryption, such as that employed in HTTPS, is insufficient to prevent a network attacker from obtaining or altering a victim’s cookies because the cookie protocol itself has various vulnerabilities (see "Weak Confidentiality" and "Weak Integrity", below). In addition, by default, cookies do not provide confidentiality or integrity from network attackers, even when used in conjunction with HTTPS.

8.2. Ambient Authority

A server that uses cookies to authenticate users can suffer security vulnerabilities because some user agents let remote parties issue HTTP requests from the user agent (e.g., via HTTP redirects or HTML forms). When issuing those requests, user agents attach cookies even if the remote party does not know the contents of the cookies, potentially letting the remote party exercise authority at an unwary server.

Although this security concern goes by a number of names (e.g., cross-site request forgery, confused deputy), the issue stems from cookies being a form of ambient authority. Cookies encourage server operators to separate designation (in the form of URLs) from authorization (in the form of cookies). Consequently, the user agent might supply the authorization for a resource designated by the attacker, possibly causing the server or its clients to undertake actions designated by the attacker as though they were authorized by the user.

Instead of using cookies for authorization, server operators might wish to consider entangling designation and authorization by treating URLs as capabilities. Instead of storing secrets in cookies, this approach stores secrets in URLs, requiring the remote entity to supply the secret itself. Although this approach is not a panacea, judicious application of these principles can lead to more robust security.

8.3. Clear Text

Unless sent over a secure channel (such as TLS [TLS13]), the information in the Cookie and Set-Cookie header fields is transmitted in the clear.

1. All sensitive information conveyed in these header fields is exposed to an eavesdropper.

2. A malicious intermediary could alter the header fields as they travel in either direction, with unpredictable results.

3. A malicious client could alter the Cookie header fields before transmission, with unpredictable results.

Servers SHOULD encrypt and sign the contents of cookies (using whatever format the server desires) when transmitting them to the user agent (even when sending the cookies over a secure channel). However, encrypting and signing cookie contents does not prevent an attacker from transplanting a cookie from one user agent to another or from replaying the cookie at a later time.

In addition to encrypting and signing the contents of every cookie, servers that require a higher level of security SHOULD use the Cookie and Set-Cookie header fields only over a secure channel. When using cookies over a secure channel, servers SHOULD set the Secure attribute (see Section 4.1.2.5) for every cookie. If a server does not set the Secure attribute, the protection provided by the secure channel will be largely moot.

For example, consider a webmail server that stores a session identifier in a cookie and is typically accessed over HTTPS. If the server does not set the Secure attribute on its cookies, an active network attacker can intercept any outbound HTTP request from the user agent and redirect that request to the webmail server over HTTP. Even if the webmail server is not listening for HTTP connections, the user agent will still include cookies in the request. The active network attacker can intercept these cookies, replay them against the server, and learn the contents of the user's email. If, instead, the server had set the Secure attribute on its cookies, the user agent would not have included the cookies in the clear-text request.

8.4. Session Identifiers

Instead of storing session information directly in a cookie (where it might be exposed to or replayed by an attacker), servers commonly store a nonce (or "session identifier") in a cookie. When the server receives an HTTP request with a nonce, the server can look up state information associated with the cookie using the nonce as a key.

Using session identifier cookies limits the damage an attacker can cause if the attacker learns the contents of a cookie because the nonce is useful only for interacting with the server (unlike non-nonce cookie content, which might itself be sensitive). Furthermore, using a single nonce prevents an attacker from "splicing" together cookie content from two interactions with the server, which could cause the server to behave unexpectedly.

Using session identifiers is not without risk. For example, the server SHOULD take care to avoid "session fixation" vulnerabilities. A session fixation attack proceeds in three steps. First, the attacker transplants a session identifier from his or her user agent to the victim's user agent. Second, the victim uses that session identifier to interact with the server, possibly imbuing the session identifier with the user's credentials or confidential information. Third, the attacker uses the session identifier to interact with server directly, possibly obtaining the user's authority or confidential information.

8.5. Weak Confidentiality

Cookies do not provide isolation by port. If a cookie is readable by a service running on one port, the cookie is also readable by a service running on another port of the same server. If a cookie is writable by a service on one port, the cookie is also writable by a service running on another port of the same server. For this reason, servers SHOULD NOT both run mutually distrusting services on different ports of the same host and use cookies to store security-

sensitive information.

Cookies do not provide isolation by scheme. Although most commonly used with the http and https schemes, the cookies for a given host might also be available to other schemes, such as ftp and gopher. Although this lack of isolation by scheme is most apparent in non-HTTP APIs that permit access to cookies (e.g., HTML's document.cookie API), the lack of isolation by scheme is actually present in requirements for processing cookies themselves (e.g., consider retrieving a URI with the gopher scheme via HTTP).

Cookies do not always provide isolation by path. Although the network-level protocol does not send cookies stored for one path to another, some user agents expose cookies via non-HTTP APIs, such as HTML's document.cookie API. Because some of these user agents (e.g., web browsers) do not isolate resources received from different paths, a resource retrieved from one path might be able to access cookies stored for another path.

8.6. Weak Integrity

Cookies do not provide integrity guarantees for sibling domains (and their subdomains). For example, consider foo.site.example and bar.site.example. The foo.site.example server can set a cookie with a Domain attribute of "site.example" (possibly overwriting an existing "site.example" cookie set by bar.site.example), and the user agent will include that cookie in HTTP requests to bar.site.example. In the worst case, bar.site.example will be unable to distinguish this cookie from a cookie it set itself. The foo.site.example server might be able to leverage this ability to mount an attack against bar.site.example.

Even though the Set-Cookie header field supports the Path attribute, the Path attribute does not provide any integrity protection because the user agent will accept an arbitrary Path attribute in a Set-Cookie header field. For example, an HTTP response to a request for http://site.example/foo/bar can set a cookie with a Path attribute of "/qux". Consequently, servers SHOULD NOT both run mutually distrusting services on different paths of the same host and use cookies to store security-sensitive information.

An active network attacker can also inject cookies into the Cookie header field sent to https://site.example/ by impersonating a response from http://site.example/ and injecting a Set-Cookie header field. The HTTPS server at site.example will be unable to distinguish these cookies from cookies that it set itself in an HTTPS response. An active network attacker might be able to leverage this ability to mount an attack against site.example even if site.example uses HTTPS exclusively.

Servers can partially mitigate these attacks by encrypting and signing the contents of their cookies, or by naming the cookie with the __Secure- prefix. However, using cryptography does not mitigate the issue completely because an attacker can replay a cookie he or she received from the authentic site.example server in the user's session, with unpredictable results.

Finally, an attacker might be able to force the user agent to delete cookies by storing a large number of cookies. Once the user agent reaches its storage limit, the user agent will be forced to evict some cookies. Servers SHOULD NOT rely upon user agents retaining cookies.

8.7. Reliance on DNS

Cookies rely upon the Domain Name System (DNS) for security. If the DNS is partially or fully compromised, the cookie protocol might fail to provide the security properties required by applications.

8.8. SameSite Cookies

SameSite cookies offer a robust defense against CSRF attack when deployed in strict mode, and when supported by the client. It is, however, prudent to ensure that this designation is not the extent of a site's defense against CSRF, as same-site navigations and submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting.

Developers are strongly encouraged to deploy the usual server-side defenses (CSRF tokens, ensuring that "safe" HTTP methods are idempotent, etc) to mitigate the risk more fully.

9. IANA Considerations

9.1. Cookie

The HTTP Field Name Registry (see [HttpFieldNameRegistry]) needs to be updated with the following registration:

Header field name: Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.5.2)

9.2. Set-Cookie

The HTTP Field Name Registry (see [HttpFieldNameRegistry]) needs to be updated with the following registration:

Header field name: Set-Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.5.1)

10. Changes

Revamped the document to allow for more detailed requirements on browsers in downstream specifications.

Acknowledgements

Many thanks to Adam Barth for laying the groundwork for a modern cookie specification with RFC 6265.

And many thanks to Steven Bingler, Lily Chen, Dylan Cutler, Steven Englehardt, Yoav Weiss, Mike West, and John Wilander for improving upon that work.

References

Normative References

[HTML] Hickson, I., Pieters, S., van Kesteren, A., J辰genstedt, P., and D. Denicola, "HTML", n.d., <https://html.spec.whatwg.org/>.

[INFRA] van Kesteren, A. and D. Denicola, "Infra", n.d., <https://infra.spec.whatwg.org>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/rfc/rfc1034>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/rfc/rfc1123>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[URL] van Kesteren, A., "URL", n.d., <https://url.spec.whatwg.org>.

Informative References

[CSRF] Barth, A., Jackson, C., and J. Mitchell, "Robust Defenses for Cross-Site Request Forgery", DOI 10.1145/1455770.1455782, ISBN 978-1-59593-810-7, ACM CCS '08: Proceedings of the 15th ACM conference on Computer and communications security (pages 75-88), October 2008, <http://portal.acm.org/citation.cfm?id=1455770.1455782>.

[HTTP] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/rfc/rfc9114>.

[HttpFieldNameRegistry] "Hypertext Transfer Protocol (HTTP) Field Name Registry", n.d., <https://www.iana.org/assignments/http-fields/>.

[RFC2109] Kristol, D. and L. Montulli, "HTTP State Management Mechanism", RFC 2109, DOI 10.17487/RFC2109, February 1997, <https://www.rfc-editor.org/rfc/rfc2109>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/rfc/rfc6265>.

[RFC7034] Ross, D. and T. Gondrom, "HTTP Header Field X-Frame-Options", RFC 7034, DOI 10.17487/RFC7034, October 2013, <https://www.rfc-editor.org/rfc/rfc7034>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9113] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/rfc/rfc9113>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

Authors' Addresses

Anne van Kesteren (editor)
Apple
Email: annevk@annevk.nl

Johann Hofmann (editor)
Google
Email: johannhof@google.com