Capitolo 3

Il protocollo HTTP

Sommario

Il concetto fondamentale che dobbiamo trattare a questo punto è il protocollo HTTP (*HyperText Transfer Protocol*), che costituisce il meccanismo di trasferimento alla base del Web e il metodo preferito per scambiare documenti indicizzati da URL tra server e client. Nonostante la presenza del termine *hypertext* nel nome, il protocollo HTTP e il vero e proprio contenuto ipertestuale (il linguaggio HTML) spesso esistono indipendentemente l'uno dall'altro. Ciò premesso, a volte sono intrecciati anche in modi sorprendenti.

La storia di HTTP offre un'interessante visione sulle ambizioni dell'autore e sulla crescente rilevanza di Internet. La prima bozza del protocollo (HTTP/0.9[1]) datata 1991 e scritta da Tim Berners-Lee, era lunga a malapena una pagina e mezza e mancava anche delle più intuitive funzioni che si sarebbero rese necessarie in futuro, come l'estendibilità richiesta per la trasmissione di dati non HTML.

Cinque anni e diverse iterazioni della specifica dopo, il primo standard ufficiale HTTP/1.0 (RFC 1945[2]) provò a rettificare queste mancanze in circa 50 pagine fitte di testo. Arriviamo al 1999 e con HTTP/1.1 (RFC 2616[3]) i sette autori accreditati tentarono di anticipare quasi ogni possibile utilizzo del protocollo, creando un volume di 150 pagine. E non è tutto: al momento in cui scriviamo, si lavora su HTTPbis[4], sostanzialmente un sostituto di HTTP/1.1, con una specifica di circa 360 pagine. Anche se molto del contenuto accumulato è irrilevante per il Web moderno, questa progressione evidenzia che il desiderio di aggiungere nuove funzioni supera di gran lunga quello di rimuovere le funzioni scarsamente utilizzate.

Oggi tutti i client e server supportano un sovrainsieme non proprio accurato di HTTP/1.0, mentre la maggioranza parla un dialetto ragionevolmente completo di HTTP/1.1, con l'aggiunta di alcune eccezioni. Nonostante non esista alcun motivo pratico per farlo, molti server web e tutti i browser più comuni mantengono la compatibilità all'indietro con HTTP/0.9.

Sintassi di base del traffico HTTP

A prima vista HTTP è un semplice protocollo di testo basato su TCP/IP. TCP (*Transmission Control Protocol*) è uno dei protocolli di comunicazione alla base di Internet. Fornisce il livello di trasporto a tutti i protocolli applicativi basati su di esso. Offre una connettività ragionevolmente affidabile, ordinata, con mutuo riconoscimento e orientata alle sessioni tra host di rete. Nella maggioranza dei casi tale protocollo si dimostra anche abbastanza resistente rispetto ad attacchi – come il blind packet spoofing – portati da altri host non locali su Internet. Ogni sessione HTTP comincia stabilendo una connessione TCP al server, tipicamente sulla porta 80, quindi effettuando una richiesta contenente un URL. In risposta, il server produce il file richiesto e, nel caso più rudimentale, chiude la connessione TCP immediatamente dopo.
Lo standard HTTP/0.9 originale non prevedeva alcuno spazio per lo scambio di metadati aggiuntivi tra le parti coinvolte. La richiesta del client consisteva sempre di un'unica riga, che inziava con GET seguito dal percorso e dalla stringa di query dell'URL e terminata con un singolo avanzamento di riga CRLF (codici ASCII 0x0D 0x0A; per i server era consigliato accettare anche un singolo LF). Una richiesta HTTP/0.9 poteva quindi avere l'aspetto seguente:

```
GET /fuzzy_bunnies.txt
```

In risposta a questo messaggio, il server inviava immediatamente il carico di dati HTML. La specifica richiedeva che i server restituissero righe lunghe esattamente 80 caratteri, ma questo consiglio non è mai stato seguito da nessuno.
L'approccio di HTTP/0.9 presenta diverse lacune sostanziali. Per esempio, non dà modo al browser di comunicare le preferenze linguistiche dell'utente, di fornire un elenco di tipologie di documenti suportati, e così via. Inoltre non dà modo al server di informare che il file richiesto non è stato trovato, che è stato spostato a un altro indirizzo o che il file scaricato non è un documento HTML – solo per iniziare.
Infine, il protocollo non è gentile con gli amministratori dei server: essendo l'URL limitato a percorso e stringa di query, è impossibile per un server ospitare siti diversi, distinti dai loro nomi di host, con un solo indirizzo IP e – a differenza dei record DNS – gli indirizzi IP non sono economici.
Per sanare queste lacune (e per fare spazio a futuri trucchi), gli standard HTTP/1.0 e HTTP/1.1 implementano un formato di conversazione leggermente differente: alla prima riga della richiesta viene aggiunta la versione del protocollo utilizzato, di seguito potranno esserci zero o più coppie *nome: valore* (note come *header*) ciascuna su una riga a sé. Gli header più comuni che si possono trovare nelle richieste sono `User-Agent` (la versione del browser), `Host` (il nome dell'host presente nell'URL), `Accept` (i tipi di documento MIME supportati. Il tipo MIME, noto anche come *Internet media type*, è un semplice valore formato da due componenti che identifica la classe e il formato di ogni

tipo di file. Il concetto nasce negli RFC 2045 e RFC 2046, dove serviva a descrivere i vari tipi di allegati della posta elettronica. L'elenco ufficiale di tali valori, come `text/plain` o `audio/mpeg`, è attualmente mantenuto dall'IANA, ma tipi *ad hoc* sono decisamente comuni), `Accept-Language` (le lingue accettate) e `Referer` (un campo – dal nome curiosamente scritto in modo errato per la lingua inglese – che indica la pagina di provenienza della richiesta, se esiste).
Gli header si concludono con un'unica riga vuota, la quale può essere seguita da qualsiasi dato il client voglia passare al server (la cui lunghezza in byte dev'essere specificata esplicitamente con l'header `Content-Length`). Il contenuto di questo carico di dati è opaco per il protocollo; in HTML questo spazio viene generalmente utilizzato per l'invio dei dati dei moduli in uno dei diversi formati possibili, anche se non è in alcun modo vincolante.
Quindi, una richiesta HTTP/1.1 può avere l'aspetto seguente:

```
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: 17
Referer: http://www.fuzzybunnies.com/main.html

I REQUEST A BUNNY
```

Il server deve rispondere a questa richiesta con una riga nella quale dichiara la versione di protocollo supportata, un codice di stato numerico (utilizzato per indicare condizioni di errore e altre circostanze particolari) e, opzionalmente, un messaggio di stato in formato leggibile dall'uomo. Quindi seguono diversi header auto-esplicativi che terminano con una riga vuota. La risposta prosegue con il contenuto della risorsa richiesta:

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

BUNNY WISH HAS BEEN GRANTED
```

Il documento RFC 2616 consente anche che la risposta venga compressa al volo utilizzando uno dei tre metodi supportati (`gzip`, `compress`, `deflate`), tranne quando il client li rifiuti esplicitamente fornendo un apposito header `Accept-Encoding`.

Le conseguenze del supporto di HTTP/0.9

Nonostante le migliorie delle versioni HTTP/1.0 e HTTP/1.1, la scomoda eredità dello "stupido" protocollo HTTP/0.9 è ancora lì, anche se normalmente è nascosta alla vista. In parte ciò è da imputare alla specifica HTTP/1.0, che richiedeva a tutti i futuri client di supportare la prima raffazzonata versione del protocollo. Nello specifico, citiamo la sezione 3.1:

> I client HTTP/1.0 devono [...] comprendere ogni valida risposta nel formato HTTP/0.9 o HTTP/1.0.

Negli anni successivi, l'RFC 2616 ha cercato di alleggerire questo requisito (sezione 19.6: "Va oltre gli scopi di una specifica di protocollo imporre la compatibilità con le versioni precedenti") ma sulla base della precedente prescrizione, tutti i browser moderni continuano a supportare il vecchio standard.
Per comprendere perché questo sia pericoloso, ricordate che i server HTTP/0.9 non rispondono con altro che il file richiesto. Non c'è alcuna indicazione del fatto che il server che risponde comprenda effettivamente l'HTTP e voglia servire un documento HTML. Tenendo presente questo, analizziamo che cosa succede se il browser invia una richiesta HTTP/1.1 a un inconsapevole servizio SMTP sulla porta 25 di `example.com`:

```
GET /<html><body><h1>Ciao! HTTP/1.1
Host: example.com:25
...
```

Dato che il server SMTP non capisce quel che sta succedendo, risponderà probabilmente in questo modo:

```
220 example.com ESMTP
500 5.5.1 Invalid command: "GET /<html><body><h1>Ciao! HTTP/1.1"
500 5.1.1 Invalid command: "Host: example.com:25"
...
421 4.4.1 Timeout
```

Tutti i browser che seguono alla lettera i documenti RFC sono obbligati ad accettare questi messaggi come il corpo di una valida risposta HTTP/0.9 e a considerarla come una pagina HTML. Questi interpreteranno il codice tra virgolette, scritto dall'aggressore, come proveniente dal legittimo sito `example.com`, e ciò interferisce profondamente con il modello di sicurezza dei browser discusso nella Parte II di questo libro e costituisce perciò un serio problema.

Trucchi con gli avanzamenti riga

Oltre ai radicali cambiamenti tra HTTP/0.9 e HTTP/1.0, sono stati messi in campo molti altri trucchi sintattici che comportano problemi di sicurezza. Cosa forse degna di nota, al contrario delle versioni che l'hanno preceduta, la specifica HTTP/1.1 richiede che i client non solo onorino gli avanzamenti riga nei formati CRLF e LF ma anche sotto forma del singolo carattere CR. Anche se questa raccomandazione è stata ignorata dai due server web più diffusi (IIS e Apache), è stata recepita sul lato client da tutti i browser tranne Firefox. La risultante inconsistenza ad aderire ha fatto sì che gli sviluppatori di applicazioni dimenticassero che non solo gli LF, ma anche i CR devono essere rimossi da ogni possibile valore controllabile da un aggressore che possa comparire in qualsiasi punto tra gli header di una connessione HTTP. Per illustrare il problema, consideriamo la seguente risposta di un server, a seguito di una richiesta in cui l'utente ha fornito dei dati che, non adeguatamente messi in sicurezza, appaiono in uno degli header, evidenziati in grassetto:

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: last_search_term=[CR][CR]<html><body><h1>Ciao![CR][LF] [CR][LF]
Action completed.
```

A Internet Explorer questa risposta apparirà come:

```
HTTP/1.1 200 OK
Set-Cookie: last_search_term=

<html><body><h1>Ciao!

Action completed.
```

Nella pratica, la classe di vulnerabilità legata alla contraffazione degli avanzamenti riga negli header HTTP – che sia dovuta all'inconsistenza dei formati o a una mancanza del filtro nell'intercettare qualche tipo di avanzamento riga particolare – è abbastanza comune da avere un proprio nome: *header injection* o *response splitting*.
Un'altra funzionalità poco conosciuta e potenzialmente pericolosa è il supporto per gli header multiriga, introdotto in HTTP/1.1. Sulla base dello standard, ogni riga di header che comincia con uno spazio bianco dev'essere considerata la continuazione della riga precedente. Per esempio:

```
X-Random-Comment: Questa è una riga molto lunga, e allora perché non va a capo
correttamente?
```

Gli header multiriga sono riconosciuti da IIS e Apache nelle richieste dei vari browser, tuttavia non sono supportati da Internet Explorer, Safari e Opera. Perciò, ogni implementazione che vi si basi o che anche solo consenta questa sintassi in una qualsiasi impostazione influenzabile da un aggressore, avrà grossi problemi. Per fortuna, casi del genere sono abbastanza rari.

Richieste tramite proxy

Molte organizzazioni e fornitori di servizi Internet fanno uso di proxy per incercettare, ispezionare e propagare richieste HTTP al posto dei propri utenti. Questo può avvenire per migliorare le prestazioni (consentendo la cache di alcune risposte frequenti del server in sistemi più vicini all'utente), per applicare regolamenti sull'uso della rete (per esempio al fine di prevenire l'accesso a siti pornografici) o per offrire accesso controllato e autenticato ad ambienti di rete altrimenti bloccati. I proxy HTTP convenzionali dipendono dal supporto esplicito del browser: l'applicazione dev'essere configurata per effettuare una richiesta modificata al sistema proxy, invece di cercare di parlare con la destinazione finale. Per scaricare una risorsa HTTP attraverso un proxy di questo tipo, il browser invierà una richiesta come questa:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
User-Agent: Bunny-Browser/1.7
Host: www.fuzzybunnies.com
...
```

La differenza fondamentale tra l'esempio precedente e la sintassi consueta è la presenza di un URL totalmente qualificato nella prima riga della richiesta (`http://www.fuzzybunnies.com/`), che indica al proxy dove collegarsi al posto dell'utente. Questa informazione è in un certo senso ridondante, dato che l'header `Host` indica già il nome dell'host; la ragione di questa duplicazione è che i due meccanismi si sono evoluti indipendentemente

l'uno dall'altro. Per evitare di essere raggirati da server e client "cospiratori", i proxy devono correggere ogni header `Host` in modo che corrisponda all'URL della richiesta o associare il contenuto nella cache con una particolare coppia URL-Host e non con uno solo dei due valori.
Molti proxy HTTP consentono ai browser di richiedere risorse non HTTP, come file o cartelle FTP. In questi casi, il proxy incapsulerà la risposta in HTTP e talvolta la convertirà in HTML quando appropriato, prima di mostrarla all'utente (in questo caso, alcuni header HTTP forniti dal client possono essere utilizzati internamente dal proxy, ma non vengono trasmessi alla destinazione finale non HTTP. Questo ha delle ricadute interessanti, se non nell'ambito della sicurezza, certamente dal punto di vista dell'ambiguità del protocollo). Detto questo, se il proxy non comprende il protocollo della richiesta, o se lo ritiene semplicemente inappropriato per potersi inserire nello scambio dei dati (per esempio nellle sessioni crittografate), deve utilizzare un approccio diverso. Un tipo speciale di richiesta, CONNECT, è riservata a questo scopo, ma non viene spiegata ulteriormente nell'RFC di HTTP/1.1. La relativa sintassi viene trattata in una specifica – solo abbozzata – del 1998[5]. Ha il seguente aspetto:

```
CONNECT www.fuzzybunnies.com:1234 HTTP/1.1
User-Agent: Bunny-Browser/1.7
...
```

Se il proxy vuole ed è in grado di collegarsi con la destinazione, accetta la richiesta con uno specifico codice di risposta HTTP e il ruolo di tale protocollo si conclude. A questo punto, il browser comincia a spedire e ricevere dati binari sul flusso TCP che si è stabilito; il proxy, da parte sua, continua a instradare il traffico tra i due punti senza interferenze.

> **NOTA**
>
> A causa di una piccola omissione nella prima versione della specifica, molti browser gestivano in modo errato i messaggi di errore non crittografati, provenienti da un proxy, in risposta al tentativo di stabilire una connessione crittografata. Tali browser interpretavano queste risposte in chiaro come se provenissero dal server di destinazione su un canale sicuro. Questa falla di fatto annullava ogni sicurezza asssociata all'utilizzo di comunicazioni cifrate sul Web. È dovuto trascorrere più di un decennio per identificare e correggere questo bug[6].

Molti altri tipi di proxy di basso livello non fanno uso del protocollo HTTP per comunicare direttamente con i browser – nonostante questo registrano i messaggi HTTP per memorizzare il contenuto nella cache o per applicare regole di sicurezza. Il tipico esempio di questo meccanismo è il proxy trasparente che intercetta silenziosamente tutto il traffico a livello TCP/IP. L'approccio dei proxy trasparenti è insolitamente pericoloso: essi possono aprire una connessione verso l'IP di destinazione e hanno a disposizione l'header `Host` della richiesta che hanno intercettato, ma non hanno alcun modo di sapere che l'indirizzo IP di destinazione sia effettivamente associato al nome server specificato. A meno di non effettuare un'ulteriore richiesta per esplicitare questa correlazione, client e server malevoli possono usare questo comportamento come campo di battaglia. Senza questi controlli aggiuntivi, all'aggressore sarebbe sufficiente collegarsi al proprio server e inviare un falso header `host: www.google.com` per far sì che il proxy memorizzi nella cache la risposta come se provenisse dal vero `www.google.com` e la presenti a tutti gli altri utenti che la chiedano.

Risoluzione di header duplicati o in conflitto

Nonostante una certa prolissità, l'RFC 2616 non approfondisce affatto le modalità con cui un browser compatibile debba risolvere potenziali ambiguità e conflitti nella richiesta o nei dati di risposta. La sezione 19.2 di questo documento RFC ("Applicazioni tolleranti") raccomanda di interpretare in modo rilassato e tollerante agli errori alcuni campi nei casi "non ambigui" – ma l'espressione stessa utilizzata nell'RFC può dirsi non ambigua?
Per esempio, in mancanza di indicazioni a livello di specifica, circa la metà dei browser preferisce la prima occorrenza di un particolare header HTTP, mentre gli altri preferiscono l'ultima. A causa di ciò, la quasi totalità delle vulnerabilità di header injection risulta sfruttabile su almeno una percentuale degli utenti obiettivo. Sul lato del server, la situazione è analogamente casuale: Apache onorerà il primo header `Host` ricevuto, mentre IIS rifiuta qualsiasi richiesta con più istanze di questo campo.
Analogamente, i documenti RFC logicamente correlati non contengono alcun divieto esplicito di mescolare header HTTP/1.0 e HTTP/1.1 potenzialmente conflittuali, né alcuna richiesta ai server o client HTTP/1.0 di ignorare del tutto la sintassi HTTP/1.1. A causa di ciò, è difficile prevedere il risultato di conflitti indiretti tra direttive HTTP/1.0 e HTTP/1.1 adibite alla stessa funzione come `Expires` e `Cache-Control`.
Infine, in alcuni rari casi, le specifiche definiscono la risoluzione dei conflitti molto chiaramente, sebbene sia assai difficile comprendere il perché si sia voluto consentire tali conflitti. Per esempio, i client HTTP/1.1 hanno l'obbligo di inviare l'header `Host` in tutte le richieste, mentre i server (e non solo i proxy) devono riconoscere anche gli URL assoluti nella prima riga delle richieste – al contrario del tradizionale metodo che usava URL relativi con i soli percorsi e stringhe di query. Questa regola permette una curiosità come questa:

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
Host: www.bunnyoutlet.com
```

In questa situazione, la sezione 5.2 dell'RFC 2616 indica che i client devono ignorare l'header Host sbagliato (ma sempre obbligatorio!) e in effetti molte implementazioni seguono questa indicazione. Il problema è che le applicazioni sottostanti spesso sono inconsapevoli di questa possibilità e possono prendere delle decisioni anche importanti sulla base dell'header errato.

NOTA

Nel lamentarci delle lacune nei documenti RFC sul protocollo HTTP, è opportuno riconoscere che le alternative possono essere altrettanto problematiche. In diversi scenari descritti negli RFC, il desiderio di normare esplicitamente la gestione di alcuni casi limite ha portato a esiti palesemente assurdi. Una di queste situazioni si è avuta con la richiesta nella sezione 3.3 dell'RFCC 1945 sulla gestione delle date in alcuni header HTTP. L'implementazione risultante (il file prtime.c nel codice sorgente di Firefox[7]) consiste di qualcosa come 2000 righe di codice C estremamente confuso e illeggibile, giusto per decifrare una data, ora e fuso orario in un modo sufficientemente a prova di errore (per impieghi quali la gestione della scadenza delle pagine nella cache).

Valori separati da punti e virgola negli header

Molti header HTTP, come `Cache-Control` o `Content-Disposition`, impiegano una sintassi con il carattere punto e virgola ";" come delimitatore, per accorpare diverse coppie *nome=valore* in un'unica riga. Il motivo per cui si permette questa notazione annidata non è chiaro; probabilmente deriva dal fatto che si ritiene un metodo più efficiente o intuitivo dell'uso di diversi header separati che avrebbero dovuto viaggiare sempre legati. In alcuni casi definiti nell'RFC2616 è permesso utilizzare stringhe racchiuse tra apici (*quoted string*) nel parametro destro di tali coppie – la sintassi è una sequenza di caratteri stampabili arbitrari racchiusa da doppi apici a fungere da delimitatori. Naturalmente il carattere doppio apice non può comparire all'interno della stringa, mentre possono comparirvi punti e virgola o spazi bianchi, cosa che consente l'invio di valori altrimenti problematici.
Sfortunatamente per gli sviluppatori, Internet Explorer non gestisce molto bene la sintassi delle stringhe racchiuse tra apici – rendendo di fatto questo schema di codifica inutilizzabile nella pratica. Il browser interpreta la seguente riga (che dovrebbe indicare che la risposta è un file scaricabile, invece di un documento in linea) in un modo inatteso:

```
Content-Disposition: attachment; filename="evil_file.exe;.txt"
```

Nell'implementazione di Microsoft il nome del file viene troncato al carattere punto e virgola e diventa, quindi, `evil_file.exe`. Questo comportamento crea un potenziale rischio per qualsiasi applicazione che esamini o aggiunga un'estensione "sicura" a un nome di file – anche se filtra adeguatamente avanzamenti riga o virgolette presenti nella stringa.

> **NOTA**
>
> Esiste un altro meccanismo per l'uso di virgolette doppie (e qualsiasi altro carattere) all'interno delle stringhe: si premette al carattere un backslash. Il meccanismo sembra essere specificato in modo errato e non è supportato da nessuno dei browser più diffusi, tranne Opera. Perché questo funzioni correttamente, i caratteri \ spuri dovrebbero essere banditi dalla stringa, cosa che non è prevista dall'RFC 2616. Il meccanismo detto *quoted-pair* permette di inserire nelle stringhe qualsiasi carattere, ivi compresi gli avanzamenti riga – cosa imcompatibile con le altre regole di interpretazione dell'HTTP.

Vale la pena notare che, quando sono presenti dei doppioni tra i campi delimitati da punti e virgola in un unico header HTTP, il loro ordine di precedenza non è definito dall'RFC. Nel caso di `filename=` in `Content-Disposition`, tutti i principali browser utilizzano la prima occorrenza trovata. In altri casi però le cose non vanno così lisce. Per esempio, estraendo il valore `URL=` dall'header `Refresh` (utilizzato per forzare il ricaricamento della pagina dopo uno specifico intervallo di tempo), Internet Explorer 6 utilizza l'ultima istanza, mentre tutti gli altri browser preferiscono la prima. E quando gestiscono `Content-Type`, Internet Explorer, Safari e Opera usano il primo valore `charset=`, mentre Firefox e Chrome usano l'ultimo.

> **NOTA**
>
> Cibo per la mente: un'affascinante rassegna di inconsistenze (generalmente non legate alla sicurezza) nella gestione del solo header HTTP `Content-Disposition` è reperibile su una pagina mantenuta da Julian Reschke: `http://greenbytes.de/tech/tc2231/`.

Set di caratteri e schemi di codifica negli header

Come i documenti che sono serviti a gettare le basi per la gestione degli URL, tutte le successive specifiche HTTP hanno bellamente evitato l'argomento di gestire caratteri non US-ASCII nei valori degli header. Esistono molti scenari in cui possono comparire caratteri non inglesi in questo contesto (per esempio nel nome del file in `Content-Disposition`), ma da qui in poi il comportamento dei browser è praticamente indefinito.
In origine, l'RFC 1945 permetteva l'uso di caratteri a 8 bit nel token TEXT (una primitiva utilizzata in vario modo per definire la sintassi di altri campi) tramite la seguente definizione:

```
OCTET   = <any 8-bit sequence of data>
CTL     = <any US-ASCII control character
          (octets 0 - 31) and DEL (127)>
TEXT    = <any OCTET except CTLs,
          but including LWS>
```

L'RFC continuava con un consiglio incomprensibile: quando si incontrano caratteri non US-ASCII in un campo TEXT, i client e i server *possono* interpretarli come ISO-8859-1, il set di caratteri standard dell'Europa Occidentale, ma non hanno l'obbligo di farlo. In seguito l'RFC 2616 ha copiato e incollato la stessa specifica, aggiungendo in una nota che le stringhe non ISO-8859-1 dovevano essere codificate utilizzando il formato definito dalla RFC 2047[8], sviluppata in origine per le comunicazioni via email.
In questo semplicissimo schema, la stringa codificata inizia con il prefisso `=?` seguito dal nome di un set di caratteri, un campo `?q?` o `?b?` per indicare il tipo di codifica (`quoted-printable` o `base64`, rispettivamente. *Quoted-printable* è un semplice schema di codifica che sostituisce ogni carattere non stampabile o comunque illegale nel contesto con il segno di uguale (=) seguito da una rappresentazione a 2 cifre esadecimali del carattere a 8 bit da codificare. Ogni eventuale segno di uguale nel testo in ingresso viene sostituito a sua volta con `=3D`. Base64 è una codifica non leggibile dall'uomo, che sostituisce stringhe di caratteri arbitrari a 8 bit con un alfabeto di caratteri alfanumerici maiuscoli e minuscoli e i segni + e /. Ogni tre byte di input producono 4 byte in output. Se l'input non finisce al limite dei 3 byte, si aggiungono uno o due segni di uguale alla fine della stringa in output) e in ultimo la stringa codificata vera e propria. La sequenza termina con `?=`. Per esempio:

```
Content-Disposition: attachment; filename="=?utf-8?q?Hi=21.txt?="
```

> **NOTA**
>
> L'RFC avrebbe anche dovuto imporre il divieto di utilizzare sequenze `=?...?=` spurie così come sono negli header, per evitare tentativi di decodifica di stringhe non intenzionalmente codificate.

Purtroppo il supporto alla codifica dell'RFC 2047 è diffuso a macchia di leopardo. È riconosciuto in alcuni header in Firefox e Chrome, mentre altri browser sono meno collaborativi. Internet Explorer sceglie di riconoscere la codifica a percentuale usata dagli URL nel campo `Content-Disposition` (abitudine adottata poi anche da Chrome) e in tal caso usa per default il set UTF-8. Firefox e Opera, d'altra parte, preferiscono

supportare una particolare sintassi di codifica a percentuale proposta nell'RFC 2231[9], solo in apparenza derivata dalla sintassi di HTTP:

```
Content-Disposition: attachment; filename*=utf-8'en-us'Hi%21.txt
```

I lettori più attenti avranno già notato che non esiste un unico schema di codifica supportato da tutti i browser. Questa situazione richiede ad alcuni sviluppatori web di ricorrere all'uso di valori a 8 bit negli header HTTP, tipicamente interpretati come UTF-8, cosa decisamente poco sicura. In Firefox per esempio, un bug presente da molto tempo fa sì che il testo UTF-8 si corrompa quando venga trasferito all'interno dell'header `Cookie`, permettendo a un cookie iniettato da un aggressore di materializzarsi in punti inattesi[10]. In altre parole, non esistono soluzioni semplici e robuste al problema.
Quando si parla di codifiche di caratteri, il problema di gestire il carattere NUL (0x00) probabilmente merita una menzione particolare. Questo carattere, usato come terminatore di stringa in molti linguaggi di programmazione, è tecnicamente vietato negli header HTTP (tranne che nel citato caso della sintassi *quoted-pair*), ma come ricorderete, agli interpreti viene richiesto di essere tolleranti. Quando questo carattere viene lasciato passare, finisce con l'avere effetti indesiderati. Per esempio, gli header `Content-Disposition` vengono troncati in corrispondenza del NUL da Internet Explorer, Firefox e Chrome, ma non da Opera o Safari.

Comportamento dell'header Referer

Come abbiamo anticipato precedentemente in questo capitolo, le richieste HTTP possono contenere un header `Referer` che contiene l'URL del documento da cui in qualche modo proviene la navigazione. Esso è destinato a favorire la ricerca di errori di programmazione e a promuovere la crescita del Web enfatizzando i riferimenti incrociati tra le pagine web.
Sfortunatamente, questo header può rivelare anche alcune informazioni sulle abitudini di navigazione dell'utente a dei malintenzionati, e lasciar scappare informazioni sensibili codificate nei parametri della stringa di query della pagina di provenienza. A causa di queste preoccupazioni e della mancanza di consigli validi su come mitigarne gli effetti, questo header viene spesso abusato per scopi di sicurezza, ma non è adatto a questo compito. Il principale problema è che non c'è modo di distinguere un client che non trasmette questo header a causa delle preferenze di privacy dell'utente, da quello che non lo trasmette per il tipo di navigazione in corso, e da quello che deliberatamente lo modifica cancellando la provenienza da un sito malevolo.
Normalmente questo header è presente nella maggior parte delle richieste HTTP e preservato anche nei reindirizzamenti HTTP, tranne nelle situazioni seguenti.

- Quando si digita manualmente un nuovo URL nella barra degli indirizzi o si apre una pagina dai preferiti o segnalibri.
- Quando la navigazione parte da un documento con un pseudo URL come `data:` o `javascript:`.
- Quando la richiesta è il risultato di un reindirizzamento controllato dall'header `Refresh` (ma non nel caso in cui si usi l'header `Location`).

- Ogni volta che il sito di provenienza è crittografato e la destinazione è in chiaro. In base alla sezione 15.1.2 dell'RFC 2616, questo avviene per motivi di privacy, ma in realtà non ha molto senso. La stringa `Referer` viene trasmessa a terzi quando si passa da un sito crittografato a un altro e, a parità di altre condizioni, l'uso della crittografia non è sinonimo di attendibilità.
- Se l'utente decide di bloccare o modificare arbitrariamente questo header agendo sulle impostazioni del browser o installando un plug-in per la privacy.

Come dovrebbe risultarvi evidente, quattro di queste cinque condizioni possono essere riprodotte di proposito da qualsiasi sito.

Tipi di richieste HTTP

La prima specifica HTTP/0.9 definiva un unico metodo (o "verb") per richiedere un documento: GET. Le successive versioni sperimentarono un sempre più strano insieme di metodi che consentissero interazioni diverse dallo scaricamento di un documento o dall'esecuzione di uno script, tra cui curiosità come SHOWMETHOD, CHECKOUT e – perché no – SPACEJUMP[11].
La maggior parte di questi esperimenti di pensiero è stata abbandonata in HTTP/1.1, che definisce un insieme molto più gestibile di otto metodi. Solo i primi due, GET e POST, hanno qualche significato per il Web di oggi.

GET

Il metodo GET è nato con il significato di "scaricamento di informazioni". In pratica è utilizzato in quasi tutte le interazioni tra client e server nel corso di una normale sessione di navigazione. Le richieste GET tradizionali non trasportano alcun dato fornito dal browser, anche se la cosa non è vietata da alcuna specifica. Questo perché le richieste GET non dovrebbero avere, per citare l'RFC, "il significato di intraprendere un'azione diversa dallo scaricamento", ovvero non dovrebbero operare modifiche persistenti allo stato di un'applicazione. Questa interpretazione ha progressivamente perso di significato nelle applicazioni moderne, dove lo stato spesso non viene nemmeno gestito sul lato del server; conseguentemente questa raccomandazione è stata ampiamente ignorata dagli sviluppatori di applicazioni (si racconta – e forse c'è un fondo di verità – di uno sfortunato webmaster di nome John Breckman, il cui sito sarebbe stato accidentalmente cancellato da un robot di un motore di ricerca. Il robot ha semplicemente scoperto un'interfaccia amministrativa non autenticata basata su GET, che John aveva scritto per il proprio sito... e ha allegramente seguito ogni collegamento "delete" che ha trovato).

NOTA

Nel protocollo HTTP/1.1, i client possono chiedere al server blocchi arbitrari (anche non contigui o sovrapposti) del documento obiettivo, specificando l'header `Range` nella richiesta GET (e, meno di frequente, in alcuni altri tipi di richiesta). Il server non deve necessariamente obbedire; ma se offre questa possibilità, i browser possono utilizzarla per riprendere lo scaricamento di file interrotti.

POST

Il metodo POST nasce per l'invio di informazioni (per lo più da moduli HTML) al server per la loro elaborazione. Dato che le azioni POST possono avere effetti persistenti sullo stato del server, molti browser chiedono conferma prima di ricaricare del contenuto scaricato con POST. Nella maggioranza dei casi, tuttavia GET e POST vengono utilizzati in maniera quasi intercambiabile.
Le richieste POST vengono spesso accompagnate da dei dati, la lunghezza dei quali è indicata dall'header `Content-Length`. Nel caso del codice HTML puro, tali dati possono consistere di stringhe con codifiche URL o MIME (un formato che vedremo in dettaglio nel Capitolo 4) anche se, di nuovo, la sintassi non è imposta in alcun modo a livello HTTP.

HEAD

HEAD è un metodo utilizzato di rado. È essenzialmente identico a GET, ma scarica solo gli header HTTP, senza i dati del documento vero e proprio. I browser in genere non effettuano richieste HEAD, ma il metodo viene a volte impiegato dai robot dei motori di ricerca e da altri strumenti automatici, per esempio per rilevare l'esistenza di un file o per controllarne la data di ultima modifica.

OPTIONS

OPTIONS è una metarichiesta che scarica l'insieme dei metodi supportati per un particolare URL (o *, che indica il server in generale) in un header di risposta. Il metodo OPTIONS non viene impiegato praticamente mai nella pratica, salvo che nell'identificazione dei server; trattandosi di informazioni dal valore limitato, possono anche non essere molto accurate.

NOTA

Per completezza dobbiamo aggiungere che le richieste OPTIONS sono uno dei cardini di uno schema di autorizzazioni tra domini e, in quanto tali, possono acquisire una certa rilevanza in futuro. Rivedremo questo schema ed esamineremo alcune funzionalità di sicurezza di prossima attivazione nei browser nel Capitolo 16.

PUT

Una richiesta PUT consente l'invio di file su un server nella posizione esatta specificata dall'URL. Dato che i browser non supportano questo metodo, le funzioni di upload dei file sono quasi sempre implementate tramite POST a script lato server, piuttosto che con questo approccio – teoricamente molto più elegante.
Detto questo, alcuni client e server HTTP non web possono utilizzare questo metodo per i loro scopi. Alcuni server web possono essere configuati (male) per accettare indiscriminatamente richieste PUT, creando un ovvio rischio per la sicurezza.

DELETE

DELETE è un metodo autoesplicativo ("cancella") complementare a PUT – e analogamente poco utilizzato nella pratica.

TRACE

TRACE è una sorta di richiesta "ping" che restituisce informazioni su tutti i proxy attraversati nell'elaborazione di una richiesta. Riporta anche l'eco della richiesta originale. I browser non fanno richieste TRACE. Tali richieste vengono utilizzate molto di rado per scopi legittimi, il primo dei quali è la verifica della sicurezza: possono rivelare interessanti dettagli sull'architettura interna dei server HTTP di una rete remota. Proprio per questa ragione, questo sistema viene quasi sempre disabilitato dagli amministratori dei server.

CONNECT

Il metodo CONNECT è concepito per stabilire connessioni non HTTP attraverso proxy HTTP. Non dev'essere inviato direttamente ai server. Se su un server viene abilitato accidentalmente il supporto al metodo CONNECT, si può avere un rischio per la sicurezza – offrire all'aggressore un modo di veicolare traffico TCP attraverso una rete altrimenti protetta.

Altri metodi HTTP

Un certo numero di altri metodi di richiesta sono poi stati sviluppati per essere impiegati da altre applicazioni o dalle estensioni dei browser; l'insieme più diffuso di tali estensioni HTTP è WebDAV, un protocollo di sviluppo e gestione delle versioni, descritto nell'RFC 4918[12].
Inoltre, anche l'API `XMLHttpRequest` consente a script JavaScript lato client di effettuare richieste con praticamente qualsiasi metodo verso il server di origine – anche se quest'ultima possibilità è fortemente ristretta in alcuni browser (lo vedremo approfonditamente nel Capitolo 9).

Codici di risposta del server

La Sezione 10 dell'RFC 2616 elenca quasi 50 codici di stato che un server può utilizzare per costruire una risposta. Circa 15 di questi vengono utilizzati nella vita reale, i rimanenti sono lì per indicare stati via via più bizzarri o improbabili come "402 Payment Required" o "415 Unsupported Media Type". La maggior parte degli stati di questo elenco non corrisponde con esattezza al comportamento delle moderne applicazioni web; la sola ragione della loro esistenza è che qualcuno ha sperato che un giorno lo avrebbe rispecchiato.
Alcuni codici meritano comunque di essere memorizzati, perché molto comuni o con significati particolari, come vediamo nel seguito.

200–299: successo

Questo intervallo di codici di stato indica il completamento di una richiesta con successo:

200 OK. È la risposta normale a una richiesta GET o POST che va a buon fine. Il browser visualizza i dati ricevuti immediatamente dopo o li processa in qualche altro modo, dipendente dal contesto.

204 No Content. Questo codice viene talvolta utilizzato per indicare una richiesta a buon fine – quando non ci si aspetta nessun dato come risposta. Il codice 204 interrompe la navigazione sull'URL che l'ha generato e mantiene l'utente sulla pagina di origine.

206 Partial Content. È analogo al codice 200. Viene prodotto dal server in risposta a richieste di blocchi da parte del browser. Il browser deve già essere in possesso delle altre parti del documento (o non ne avrebbe richiesto solo un blocco) ed esaminerà l'header di risposta `Content-Range` per ricomporlo prima di effettuarne l'elaborazione.

300–399: reindirizzamento e altri messaggi di stato

Questi codici vengono utilizzati per comunicare una varietà di stati che non indicano un errore, ma che richiedono un trattamento particolare da parte del browser:

301 Moved Permanently, 302 Found, 303 See Other. Queste risposte indicano al browser di ripetere la richiesta a un nuovo server, specificato nell'header di risposta `Location`. Nonostante le distinzioni che vengono fatte nell'RFC, tutti i browser moderni quando incontrano questi codici di risposta rimpiazzano POST con GET, eliminano il payload e reinoltrano la richiesta automaticamente.

> **NOTA**
>
> I messaggi reindirizzati possono contenere un payload. In tal caso, tali messaggi non vengono visualizzati all'utente a meno che il reindirizzamento non risulti possibile (per esempio nel caso di un valore *`Location`* mancante o non supportato). In alcuni browser, poi, questi messaggi possono essere soppressi anche in scenari di questo tipo.

304 Not Modified. Questa risposta indiretta informa il client che il documento richiesto non è stato modificato rispetto alla copia già in suo possesso. È la risposta a una richiesta condizionata con header come `If-Modified-Since`, che viene inviata per convalidare nuovamente la cache di documenti del browser. Il corpo della risposta non viene visualizzato all'utente. Se il server risponde in questo modo a una richiesta non condizionata, il risultato dipende dal browser e può essere ridicolo; per esempio, Opera apre una richiesta di download che poi non funziona.

307 Temporary Redirect. Analogamente al codice 302, ma a differenza degli altri metodi di reindirizzamento, i browser non trasformano la richiesta da POST a GET per eseguire un reindirizzamento 307. Questo codice non si usa spesso nelle applicazioni web, e alcuni browser non si comportano coerentemente quando lo ricevono.

400–499: errori lato client

Questo intervallo di valori viene utilizzato per indicare condizioni di errore provocate dal comportamento del client:

400 Bad Request (e messaggi correlati). Il server non è in grado o non vuole processare la richiesta per qualche motivo non specificato. Il payload della risposta in genere spiega il problema e generalmente viene gestito dal browser come una risposta 200. Esistono anche varianti più specifiche, come "411 Length Required", "405 Method Not Allowed" o "414 Request-URI Too Long". Ci chiediamo tutti perché l'assenza dell'header `Content-Length` meriti un codice di risposta dedicato (411), mentre l'assenza dell'header `Host` debba accontentarsi solo di un codice 400 generico.

401 Unauthorized. Questa risposta indica che l'utente deve fornire delle credenziali di autenticazione a livello di protocollo HTTP per accedere alla risorsa. In genere a questo punto il browser chiede all'utente le informazioni per fare il login e visualizza il corpo della risposta solo se il processo di autenticazione non va a buon fine. Questo meccanismo viene illustrato in maggior dettaglio più avanti nel paragrafo dedicato all'autenticazione HTTP.

403 Forbidden. L'URL richiesto esiste ma non può essere scaricato per ragioni diverse dall'autenticazione HTTP, per esempio permessi di accesso al filesystem non sufficienti, una regola di configurazione sul server che vieta l'accesso alla risorsa, credenziali insufficienti di qualche tipo – come cookie non validi o un indirizzo IP di origine non valido. Questa risposta viene generalmente visualizzata all'utente.

404 Not Found. L'URL richiesto non esiste. Questa risposta viene generalmente mostrata all'utente.

500–599: errore lato server

È una classe di messaggi di errore, prodotti come risposta a problemi verificatisi sul server:

500 Internal Server Error, 503 Service Unavailable, e così via. Il server ha un problema che determina l'impossibilità di onorare la richiesta. Può trattarsi di un problema temporaneo, il risultato di una configurazione errata o semplicemente l'effetto di una richiesta a una locazione inattesa. Questa risposta viene generalmente mostrata all'utente.

Coerenza dei codici d'errore HTTP

Dato che non c'è alcuna differenza immediatamente osservabile nel restituire la maggioranza dei codici 2xx, 4xx e 5xx, tali valori non vengono scelti con alcun particolare zelo. In particolare, le applicazioni web sono note per la loro abitudine di restituire "200 OK" anche quando si è verificato un errore – la cui comunicazione avviene sulla pagina risultante (è uno dei tanti fattori che hanno reso il test automatico delle applicazioni web assai più complesso di quanto dovrebbe).

In rare occasioni vengono inventati nuovi – e non necessariamente appropriati – codici HTTP per usi specifici. Alcuni di questi sono standardizzati, come un paio introdotti dall'RFC di WebDAV[13]. Altri invece, come la risposta di stato "449 Retry With", non lo sono.

Sessioni keepalive

In origine le sessioni HTTP erano progettate per essere aperte e chiuse immediatamente: effettuare una richiesta per ciascuna connessione TCP, ottenere il dato, ripetere. Il sovraccarico di ripetere continuamente una negoziazione TCP a tre passaggi (e di creare un nuovo processo con un fork nel modello tradizionale Unix) ha presto evidenziato un collo di bottiglia prestazionale, così nel protocollo HTTP/1.1 si è introdotta il concetto delle sessioni mantenute aperte (*keepalive*).
Il protocollo esistente dava già al server l'indicazione di dove terminasse la richiesta dell'utente (una riga vuota, seguita opzionalmente da `Content-Length` byte di dati), ma per continuare a usare la connessione esistente, anche il client doveva sapere di quanti byte fosse costituito il documento scaricato; in questo modo la chiusura della connessione non sarebbe più servita come indicatore. Perciò, le sessioni keepalive richiedono solo che la risposta contenga a sua volta un header `Content-Length` che specifichi la quantità di dati che seguiranno. Una volta che il client ha ricevuto il numero di byte preannunciati, sa che può procedere a effettuare una seconda richiesta e mettersi in attesa di una nuova risposta.
Anche se è molto migliorativo sotto il punto di vista prestazionale, questo meccanismo esacerba l'impatto dei bug di delimitazione nelle richieste e nelle risposte HTTP. È fin troppo facile portare fuori sincrono il client o il server su una richiesta o una risposta. Per illustrare questo concetto, consideriamo un server che pensi di inviare una sola risposta HTTP, strutturata come segue:

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: term=[CR]Content-Length: 0[CR][CR]HTTP/1.1 200 OK[CR]Gotcha: Yup[CR][LF]
Content-Length: 17[CR][LF]
[CR][LF]
Action completed.
```

Il client, invece, può vedere due risposte e associare la prima alla propria richiesta e la seconda a una query ancora da inviare (in teoria i client possono essere progettati in modo da cestinare ogni eventuale dato non richiesto che arrivi dal server prima di inviare nuove richieste in una sessione keepalive, di fatto limitando l'impatto di un attacco. Questo non avviene nella pratica per via del cosiddetto *HTTP pipelining*: per motivi di prestazioni, alcuni client sono progettati per evadere molte richieste nello stesso momento, senza attendere il completamento di quelle pendenti), che può essere indirizzata addirittura a un nome di host differente sullo stesso IP:

```
HTTP/1.1 200 OK Set-Cookie: term= Content-Length: 0

HTTP/1.1 200 OK
Gotcha: Yup
Content-Length: 17

Action completed.
```

Se poi la risposta viene vista da un proxy HTTP di cache, il risultato sbagliato può essere addirittura memorizzato globalmente e servito ad altri utenti: cosa davvero pessima per la sicurezza.
Un sistema molto più sicuro per le sessioni keepalive dovrebbe specificare sia la lunghezza degli header che del payload, o utilizzare un delimitatore generato casualmente volta per volta per delimitare ogni risposta. Purtroppo non è stato previsto niente del genere.
Le sessioni keepalive sono utilizzate per default in HTTP/1.1 a meno che non vengano esplicitamente inibite (`Connection: close`) e sono supportate da molti server HTTP/1.0 quando incontrano l'header `Connection: keep-alive`. Sia i server che i browser possono limitare il numero di richieste concorrenti per ciascuna connessione e specificare il massimo intervallo di tempo per cui una connessione inattiva viene mantenuta aperta

Trasferimenti di dati a blocchi

Una significativa limitazione delle sessioni keepalive basate su `Content-Length` è che richiedono che il server conosca in anticipo l'esatta dimensione in byte della risposta. Se questo è molto semplice nel caso di file statici, perché l'informazione è già disponibile al filesystem, la cosa si complica di molto quando la risposta sia generata dinamicamente, perché la pagina dev'essere memorizzata in cache interamente prima che possa essere inviata al client. La sfida diventa insormontabile se il payload è molto grande o se viene prodotto gradualmente (pensate allo streaming video). In questi casi la memorizzazione anticipata per poter calcolare la dimensione della risposta è assolutamente fuori questione.
In risposta a questo problema, la Sezione 3.6.1 dell'RFC 2616 fornisce ai server la possibilità di utilizzare lo schema `Transfer-Encoding: chunked`, in cui il payload viene trasferito in blocchi a mano a mano che si renda disponibile.
La lunghezza di ciascuna porzione del documento viene dichiarata all'inizio mediante un numero esadecimale su una riga a parte. La lunghezza totale del documento è indeterminata e lo scaricamento continua finché non viene rilevato un blocco finale con lunghezza zero.
Un esempio di risposta a blocchi ha l'aspetto seguente:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
...

5
Hello
6 world!
0
```

Non ci sono problemi significativi nel supporto ai trasferimenti a blocchi, se non forse la possibilità che blocchi patologicamente grossi provochino degli overflow nel codice del browser o richiedano di risolvere delle incongruenze tra `Content-Length` e la lunghezza dei blocchi stessi (la specifica dà la precedenza alla lunghezza dei blocchi, anche se ogni tentativo di risolvere elegantemente questa particolare situazione appare imprudente). Tutti i principali browser gestiscono queste condizioni in modo corretto, ma le nuove implementazioni devono guardarsi bene alle spalle.

Comportamento della cache

Per motivi legati alle prestazioni e all'ottimizzazione dell'uso della banda, i client HTTP e alcuni intermediari utilizzano la cache per memorizzare le risposte HTTP e riutilizzarle in seguito. Questo dev'essere sembrato un compito semplice nei primi giorni del Web, ma si è caricato di insidie a mano a mano che sulla rete si sono inseriti dati sempre più sensibili degli utenti, e tali dati sono stati aggiornati sempre più frequentemente.
La Sezione 13.4 dell'RFC 2616 consiglia che le richieste GET che ottengono in risposta alcuni codici HTTP (come "200 OK" e "301 Moved Permanently") possano essere implicitamente poste nella cache in assenza di altre direttive fornite dal server. Tale risposta può essere archiviata nella cache indefinitamente e può essere riutilizzata per ogni futura richiesta che abbia lo stesso metodo e URL di destinazione, anche se vi sono differenze in altri parametri (come gli header `Cookie`). C'è invece il divieto di memorizzare richieste che fanno uso di autenticazione HTTP (cfr. il paragrafo dedicato all'autenticazione HTTP), mentre altri metodi di autenticazione, come i cookie, non sono contemplati nella specifica.
Quando una risposta viene memorizzata nella cache, l'implementazione può scegliere di convalidarla prima di riutilizzarla – ma la maggior parte delle volte questo non è richiesto. La ri-validazione avviene tramite una richiesta con un particolare header condizionato, come `If-Modified-Since` (seguito dalla data di memorizzazione della risposta precedente) o `If-None-Match` (seguito da un header `ETag` copiato così come fornito dal server assieme alla copia della pagina memorizzata). Il server può rispondere con il codice HTTP "304 Not Modified" o con una nuova copia della risorsa.

> **NOTA**
>
> Le coppie di header `Date/If-Modified-Since` ed `ETag/If-None-Match` insieme a `Cache-Control: private` costituiscono un metodo comodo e totalmente imprevisto di memorizzare token univoci e a lunga scadenza nel browser[14]. Lo stesso risultato si può ottenere anche depositando un token univoco in un file JavaScript memorizzabile nella cache e restituendo il codice "304 Not Modified" a ogni successiva richiesta condizionale che punti alla pagina che genera il token. A differenza dei meccanismi scritti ad hoc come i cookie HTTP (discussi nel paragrafo seguente), gli utenti hanno ben poco controllo sulle informazioni memorizzate nella cache del browser, sulle relative condizioni e sul periodo per cui rimangono memorizzate.

Il sistema di cache implicito è decisamente problematico. Per questo motivo i server dovrebbero sempre utilizzare direttive di caching esplicite. Per venire in soccorso, la specifica HTTP/1.0 fornisce un header apposito `Expires` che indica la data dopo la quale una copia cache dev'essere eliminata; se questo valore è uguale all'header `Date` fornito dal server, la risposta non è archiviabile nella cache.
Oltre a questa regola banale, la connessione tra `Expires` e `Date` non viene specificata: non è chiaro se `Expires` debba essere confrontato con l'orologio di sistema sul sistema della cache (cosa problematica se gli orologi di server e client non sono sincronizzati) o valutato sulla base della differenza tra `Expires` e `Date` (approccio più robusto ma che può incepparsi se `Date` viene accidentalmente omesso). Firefox e Opera usano la seconda interpretazione, mentre tuti gli altri preferiscono la prima.

Nella maggioranza dei browser, un valore errato di `Expires` disabilita la cache, ma fare affidamento su questo meccanismo è un azzardo.
I client HTTP/1.0 possono utilizzare anche l'header di richiesta `Pragma: no-cache`, che dev'essere interpretato dal proxy come la volontà di ottenere una nuova copia della risorsa richiesta, invece di una copia memorizzata. Alcuni proxy HTTP/1.0 riconoscono anche un header di risposta `Pragma: no-cache` fuori standard come una richiesta di non memorizzare una copia del documento in transito.
Per contrasto, HTTP/1.1 utilizza un approccio decisamente più completo alle direttive per la cache e introduce un nuovo header `Cache-Control`. Tale header può assumere valori quali `public` (il documento è memorizzabile pubblicamente), `private` (i proxy non hanno il permesso di memorizzare), `no-cache` (che è un po' confuso – la risposta può essere memorizzata ma non dev'essere riutilizzata per richieste future – l'RFC è poco chiara in merito, ma sembra che lo scopo sia quello di permettere l'uso del documento memorizzato solo per poter navigare "indietro" e "avanti" con i pulsanti del browser e non quando è necesario un vero e proprio refresh della pagina. Firefox segue questa interpretazione, tutti gli altri browser invece considerano `no-cache` e `no-store` come equivalenti) e `no-store` (assolutamente mai memorizzare).
Le direttive `public` e `private` possono essere accompagnate da un qualificatore come `max-age`, che specifica il tempo massimo per cui mantenere una copia, o `must-revalidate`, che impone l'effettuazione di una richiesta condizionale prima di poter riutilizzare il contenuto.
Sfortunatamente, in genere è necessario che i server restituiscano le direttive di cache sia in formato HTTP/1.0 che in formato HTTP/1.1, perché alcuni tipi di proxy commerciali non interpretano `Cache-Control` correttamente. Per evitare con certezza che una pagina venga messa in cache è necessario usare questo insieme di header di risposta:

```
Expires: [current date]
Date: [current date]
Pragma: no-cache
Cache-Control: no-cache, no-store
```

Quando queste direttive di cache sono discordanti, il risultato è difficile da prevedere. Alcuni browser favoriscono le direttive HTTP/1.1 e danno la precedenza a `no-cache`, anche se viene seguito erroneamente da `public`; altri no.
Un altro rischio della cache HTTP si ha sulle reti non sicure, come le reti Wi-Fi pubbliche, che consentono a un aggressore di intercettare richieste a un URL e restituire all'utente contenuti modificati e a lunga data di scadenza.
Se poi la cache "avvelenata" del browser viene riutilizzata su una rete fidata, il contenuto iniettato tornerà in superficie inaspettatamente.
E per di più la vittima non deve nemmeno aver visitato l'applicazione obiettivo: è possibile iniettare un riferimento a un dominio sensibile scelto con cura dall'aggressore in un altro contesto.
Non esistono ancora buone soluzioni a questo problema; comunque è sempre utile ripulire la cache del browser dopo essere stati in una biblioteca con area Wi-Fi!

Semantica dei cookie HTTP

I cookie HTTP non fanno parte dell'RFC 2616, ma sono una delle principali estensioni del protocollo utilizzate sul Web. Il meccanismo dei cookie consente ai server di memorizzare minuscole coppie *nome=valore* in modo trasparente nel browser, inviando l'header di risposta `Set-Cookie` e ricevendole indietro nelle future richieste tramite il parametro client `Cookie`. I cookie sono decisamente il modo più usato di mantenere le sessioni aperte e autenticare le richieste degli utenti; sono una delle quattro forme canoniche di *ambient authority* sul Web; le altre sono l'autenticazione integrata HTTP, la verifica degli IP e i certificati dei client. L'ambient authority è una forma di controllo d'accesso basata su una proprietà globale e persistente dell'entità richiedente, invece che su una forma esplicita di autorizzazione che sarebbe valida solo per un'azione specifica. Un cookie che identifichi un utente, che venga incluso indiscrimitatamente in ogni richiesta in uscita verso un dato server, senza alcuna considerazione sul perché venga fatta tale richiesta, ricade in questa categoria.

Implementato in origine da Lou Montulli in Netscape intorno al 1994, e descritto in un breve documento informale di quattro pagine[15], questo meccanismo non è stato inserito in alcuno standard nei successivi 17 anni. Nel 1997 l'RFC 2109[16] ha cercato di documentare questo status quo ma, inspiegabilmente, ha proposto un certo numero di modifiche che, a oggi, la renderebbero sostanzialmente incompatibile con l'attuale comportamento dei browser moderni. Un altro ambizioso sforzo – *Cookie2* – venne alla luce nell'RFC 2965[17] ma un decennio più tardi continua a non avere il supporto di alcun browser, situazione che ha ben poche probabilità di cambiamento. Un nuovo sforzo per scrivere una specifica ragionevolmente accurata sul meccanismo dei cookie – RFC 6265[18] – si è concluso appena prima della pubblicazione di questo libro, ponendo finalmente termine a questa mancanza di certezze.

A causa della prolungata assenza di standard reali, le implementazioni si sono evolute in modi molto interessanti e talvolta incompatibili. In pratica, si possono creare dei nuovi cookie utilizzando l'header `Set-Cookie` seguito da un'unica coppia *nome=valore* e da un certo numero di parametri opzionali separati da punto e virgola che ne definiscono campo d'applicazione e validità.

- `Expires`. Specifica la data di scadenza di un cookie in un formato simile a quello usato per gli header `Date` ed `Expires`. Se un cookie viene creato senza un'esplicita data di scadenza, viene generalmente tenuto in memoria per la durata della sessione del browser (che, specialmente sui computer portatili con la funzione di sospensione, può anche significare settimane). I cookie con scadenza regolare, invece, vengono salvati su disco e sopravvivono tra le varie sessioni, a meno che le impostazioni sulla privacy dell'utente non ne impongano la cancellazione.
- `Max-age`. Questo metodo alternativo di impostare una data di scadenza – suggerito dai documenti RFC – non è supportato da Internet Explorer e perciò non viene utilizzato nella pratica.
- `Domain`. Questo parametro permette al cookie di essere utilizzato in un dominio più vasto del nome di host restituito dall'header `Set-Cookie`. Le regole precise di questo meccanismo e le relative conseguenze sulla sicurezza sono trattate nel Capitolo 9.

NOTA

Contrariamente a quanto viene detto nell'RFC 2109, non è possibile estendere la validità di un cookie a uno specifico nome di host utilizzando questo parametro. Per esempio, `domain=example.com` ricomprenderà sempre anche `www.example.com`. L'unico modo di creare cookie con un unico host di validità consiste nell'omettere `domain`, ma anche questo approccio non funziona come ci si aspetterebbe in Internet Explorer.

- `Path`. Consente a un cookie di avere validità a partire da un particolare percorso. Non è un meccanismo di sicurezza attuabile nella pratica per le ragioni che vedremo nel Capitolo 9. Può essere utilizzato per comodità, per evitare che dei cookie con nomi identici possano essere utilizzati in diverse parti di una stessa applicazione.
- `Secure`. Vieta che il cookie venga trasmesso su connessioni non crittografate.
- `HttpOnly`. Fa sì che il cookie non possa essere letto attraverso l'API `document.cookie` di JavaScript. È un'estensione di Microsoft che però oggi è supportata da tutti i maggiori browser.

Nell'effettuare le successive richieste a un dominio per cui hanno dei cookie validi, i browser combinano tutte le coppie *nome=valore* in un unico header `Cookie` separato da punti e virgola, senza alcun metadato aggiuntivo, che spediscono al server. Se ci sono troppi cookie da inviare in una particolare richiesta, vengono superati i limiti dimensionali per gli header sul server e la richiesta fallisce; non c'è alcun modo per risolvere questa condizione, se non quella di ripulire manualmente l'archivio dei cookie.
Curiosamente, non esiste alcun metodo esplicito con cui i server HTTP possano richiedere la cancellazione dei cookie non più necessari. Tuttavia, ciascun cookie è identificato univocamente da una tupla nome-dominio-percorso (gli attributi `secure` e `httponly` vengono ignorati), e questo permette a un vecchio cookie con lo stesso campo di applicazione di essere semplicemente sovrascritto. Inoltre, se il cookie che sovrascrive ha una data di scadenza già trascorsa (la data di `expires` è collocata nel passato), verrà immediatamente azzerato: un modo bizantino di cancellare dei dati.
Anche se l'RFC 2109 richiede che debbano essere accettati diversi valori separati da virgola in un unico header `Set-Cookie`, questa pratica è pericolosa e non più supportata dai browser. Firefox permette di impostare più cookie per volta tramite l'API Javascript `document.cookie` ma, inspiegabilmente, richiede degli avanzamenti riga come delimitatori al posto delle virgole. Nessun browser usa le virgole come delimitatori dei `Cookie`, e riconoscerle lato server andrebbe considerato poco sicuro.
Un'altra importante differenza tra la specifica e la realtà consiste nel fatto che i valori nei cookie dovrebbero utilizzare il formato quoted-string delle specifiche HTTP (cfr. il precedente paragrafo dedicato agli header delimitati da punto e virgola), ma solo Firefox e Opera riconoscono tale sintassi nella pratica. Affidarsi a valori quoted-string è quindi insicuro, così come permettere virgolette spurie all'interno di cookie controllati da malintenzionati.
I cookie non garantiscono di essere particolarmente affidabili. I programmi lato utente permettono di influire ben poco sul numero e la dimensione dei cookie permessi per ciascun dominio e, come caratteristica male interpretata di privacy, possono limitarne la durata di vita. Dato che si può ottenere la tracciatura dell'utente con affidabilità equivalente in altri modi, gli sforzi di applicare la tracciatura basata sui cookie producono più danno che benefici.

Autenticazione HTTP

L'autenticazione HTTP, come specificato nell'RFC 2617[19], è il meccanismo originario per la gestione delle credenziali nelle applicazioni web – oggi quasi completamente estinto. Le ragioni di questo fallimento sono state l'inflessibilità dell'interfaccia utente associata a questo meccanismo nel browser, la difficoltà di incudervi schemi con credenziali più sofisticate di una password o l'impossibilità di controllare per quanto tempo le credenziali vengano memorizzate e con quali altri domini vengano condivise.

In ogni caso, lo schema di base è decisamente semplice. Tutto comincia col browser che fa una richiesta non autenticata, a cui il server risponde "401 Unauthorized" (i termini *autenticazione* e *autorizzazione* vengono utilizzati in modo intercambiabile in questo documento RFC, ma hanno significati distinti nell'ambito della sicurezza dell'informazione. Con autenticazione si intende il processo di provare la propria identità, mentre l'autorizzazione è il processo di determinare se le credenziali ottenute permettono di svolgere una specifica azione). Il server invia anche un header `WWW-Authenticate` in cui specifica il metodo di autenticazione richiesto, il regno di autenticazione detto *realm* (un identificatore arbitrario a cui vengono legate le credenziali fornite) e altri parametri specifici del metodo, se applicabili.

Il client deve quindi ottenere le credenziali in un modo o nell'altro, codificarle nell'header `Authorization` e inviare nuovamente la richiesta originale con tale header. In base a questa specifica, per motivi prestazionali, lo stesso header `Authorization` può essere incluso nelle successive richieste allo stesso percorso sul server senza un secondo challenge `WWW-Authenticate`. È consentito riutilizzare le stesse credenziali in risposta ad altri challenge `WWW-Authenticate` altrove sul server, se la stringa `realm` e il metodo di autenticazione sono gli stessi.

In pratica, questo consiglio non viene seguito con molta precisione: tranne Safari e Chrome, gli altri browser ignorano la stringa `realm` o hanno un approccio molto rilassato nella verifica dei percorsi. Tutti i browser, invece, restringono l'uso delle credenziali memorizzate non solo al server di destinazione, ma anche allo specifico protocollo e porta utilizzati per la connessione, cosa che offre evidenti vantaggi in termini di sicurezza.

I due metodi di passaggio delle credenziali specificati nell'RFC originale sono `basic` e `digest`. Il primo essenzialmente invia la password in chiaro, codificata in base64. L'altro calcola un hash crittografico monouso che protegge la password dalla visualizzazione in chiaro ed evita che possa essere riutilizzato l'header `Authorization` in una successiva richiesta. Sfortunatamente, i browser moderni supportano entrambi i metodi e non fanno distinzione tra i due. Risultato: gli aggressori possono semplicemente sostituire la parola `digest` con `basic` nella richiesta iniziale per ottenere una password in chiaro non appena l'utente invia le credenziali. Sorprendentemente, la Sezione 4.8 dell'RFC prevedeva questo rischio e offriva un consiglio tanto utile quanto ignorato:

> I programmi utente devono considerare all'atto dell'autenticazione misure quali un'indicazione visiva del meccanismo di trasferimento impiegato oppure ricordare lo schema di autenticazione più forte utilizzato da un server e produrre un avvertimento prima di passare a uno più debole. Può essere una buona idea che il programma utente richieda l'autenticazione digest come default o sempre verso alcuni siti specifici.

In aggiunta a questi due metodi di autenticazione, alcuni browser supportano anche metodi meno comuni, come `NTLM` e `Negotiate` di Microsoft, utilizzati in maniera trasparente con le credenziali di accesso ai domini Windows[20].
Anche se l'autenticazione HTTP si incontra poco di frequente su Internet, continua a proiettare una certa ombra su alcuni tipi di applicazioni web. Per esempio, quando un'immagine esterna fornita da un aggressore viene inserita in un argomento su un forum di discussione e il server che la ospita decide di colpo di restituire un "401 Unauthorized" ad alcune richieste, ai lettori di quell'articolo si presenterà la finestra con la richiesta di password. Dopo aver controllato la barra degli indirizzi, molti confonderanno tale richiesta con quella di inserire le credenziali di accesso al forum – che verranno immediatamente spedite al server che ospita l'immagine dell'aggressore. Oops!

Crittografia a livello di protocollo e certificati client

Come dovrebbe risultare ormai evidente, tutte le informazioni delle sessioni HTTP vengono scambiate come testo in chiaro nella rete. Negli anni Novanta questo non sarebbe stato un gran problema: certo il testo in chiaro esponeva i propri gusti di navigazione alla vista di provider di Internet impiccioni, o forse ad altri utenti maliziosi del proprio ufficio, o a qualche zelante agenzia governativa, ma la cosa non sembrava peggiore del comportamento di SMTP, DNS e di tutti gli altri protocolli di uso comune. Putroppo la crescita di popolarità del Web come piattaforma di commercio elettronico ha aggravato il rischio, e la sostanziale regressione della sicurezza di rete causata dalla diffusione di reti wireless pubbliche è stata la goccia che ha fatto traboccare il vaso.
Dopo alcune pezze di scarso successo, una vera soluzione a questo problema fu proposta nell'RFC 2818[21]: perché non incapsulare normali richieste HTTP all'interno di un esistente, multifunzionale meccanismo di sicurezza noto come TLS o SSL (*Transport Layer Security*) sviluppato diversi anni prima? Questo sistema di trasporto fa uno di crittografia a chiave pubblica per creare un canale di comunicazione confidenziale e autenticato tra due punti di rete, senza richiedere alcun tipo di modifica al protocollo HTTP. La *crittografia a chiave pubblica* si basa su algoritmi di crittografia asimmetrici. Viene creata una coppia di chiavi: una privata, mantenuta segreta dal proprietario e necessaria per decifrare i messaggi, e una pubblica, che viene diffusa al mondo e utile solo per cifrare i messaggi destinati al proprietario (ma non per decifrarli).
Per consentire ai server web di provare la propria identità, ogni browser compatibile con HTTPS viene fornito con un piccolo corredo di chiavi pubbliche appartenenti ad alcune autorità di certificazione (CA, *Certificate Authorities*). Le CA sono organizzazioni ritenute affidabili dai fornitori di browser, che attestano che una particolare chiave pubblica appartiene a uno specifico sito – dopo aver convalidato l'identità della persona che richiede tale attestazione e averne verificato la titolarità del dominio in questione.
L'insieme delle organizzazioni di fiducia è vario, arbitrario e non particolarmente ben documentato, cosa che spesso solleva motivate critiche. Nonostante tutto, alla fine il sistema fa il proprio lavoro ragionevolmente bene. Solo alcuni svarioni sono stati documentati a oggi (compresa la recente compromissione di un'azienda di alto profilo di nome Comodo[22]), e non ci sono notizie di abusi dei privilegi di CA.
Nell'implementazione attuale, quando si stabilisce una nuova connessione HTTPS il browser riceve una chiave pubblica firmata dal server; ne verifica la firma (che non può

essere applicata se non si ha accesso alla chiave privata della CA), poi controlla che i campi `cn` (*common name*) o *subjectAltName* del certificato indichino che è stato rilasciato per il server con cui si vuole dialogare e si accerta che la chiave non sia inserita in un elenco pubblico di chiavi revocate (per esempio perché compromessa o ottenuta con metodi illegali). Se l'esito di tutte queste verifiche è positivo, il browser può procedere a crittografare i messaggi diretti al server con la chiave pubblica che ha ricevuto ed essere certo che solo il server potrà decifrarli.
Normalmente il client rimane anonimo: genera una chiave crittografica temporanea, ma il processo non prova l'identità del client. Una prova di questo tipo può essere necessaria, però. In alcune organizzazioni si è iniziato a utilizzare i certificati per i client e in molte nazioni sono stati adottati addirittura a livello statale (per esempio per meccanismi di e-government). Dato che lo scopo principale dei certificati client è quello di fornire informazioni sull'identità reale dell'utente, in genere i browser chiedono conferma prima di inviarli a siti non conosciuti, per ragioni di privacy; oltre a questo, il certificato può fungere come una ulteriore forma di ambient authority.
Vale la pena di notare che, sebbene HTTPS in quanto tale sia uno schema in grado di resistere ad attacchi sia attivi che passivi, non fa molto per nascondere l'evidenza di un accesso a informazioni pubbliche. Non maschera la richiesta HTTP grezza e le dimensioni delle risposte, la direzione del traffico e le tempistiche di una tipica sessione di navigazione. La cosa dà la possibilità a un aggressore passivo e con pochissima padronanza tecnica di venire a conoscenza, per esempio, di quale imbarazzante pagina di Wikipedia la vittima stia visualizzando su un canale cifrato. Infatti, in un caso estremo, alcuni ricercatori Microsoft hanno illustrato l'uso di sistemi di profilazione dei pacchetti e hanno ricostruito la sequenza di tasti digitati dall'utente durante l'uso di un'applicazione online[23].

Certificati EV SSL

Nei primi tempi dell'HTTPS, molte autorità di certificazione pubbliche effettuavano accurati e approfonditi controlli di identità e di possesso dei domini prima di firmare un certificato. Sfortunatamente, in ossequio alla riduzione dei costi, alcune CA si accontentano di una carta di credito valida e della capacità di mettere un file su un server per completare il processo di verifica. Questo approccio rende praticamente tutti i campi di un certificato che non siano `cn` e `subjectAltName` non degni di fiducia.
Per risolvere il problema è stato introdotto un nuovo tipo di certificato, marcato con uno speciale flag e venduto a un prezzo significativamente più elevato: l'EV SSL (*Extended Validation SSL*). Questi certificati non solo provano il possesso del dominio, ma attestano anche l'identità del richiedente, a seguito di una verifica manuale. I certificati EV SSL sono riconosciuti da tutti i browser moderni che, in risposta, rendono verde o blu una parte della barra degli indirizzi. Sebbene il beneficio di tali certificati sia fuori questione, l'idea di associare certificati a prezzo più elevato a una segnalazione dal significato di "maggior livello di sicurezza" è spesso criticato come un nuovo modo ingegnoso di ricavare profitto dalla stessa tecnologia.

Regole di gestione degli errori

In un mondo ideale, un sospetto di errore in un certificato SSL, come un nome di host scritto in modo fuorviante o un'autorità di certificazione non riconosciuta, causerebbe l'impossibilità di stabilire una connessione. Errori meno sospetti, come certificati scaduti da pochi giorni o un nome di host errato, potrebbero essere accompagnati da semplici avvisi.

Sfortunatamente la maggioranza dei browser ha indiscriminatamente scaricato la responsabilità del problema sull'utente, cercando di spiegare la crittografia in termini da profani (fallendo clamorosamente) e richiedendo che fosse quest'ultimo a prendere la decisione: vuoi veramente vedere questa pagina o no? La Figura 3.1 mostra una finestra di dialogo di questo tipo.

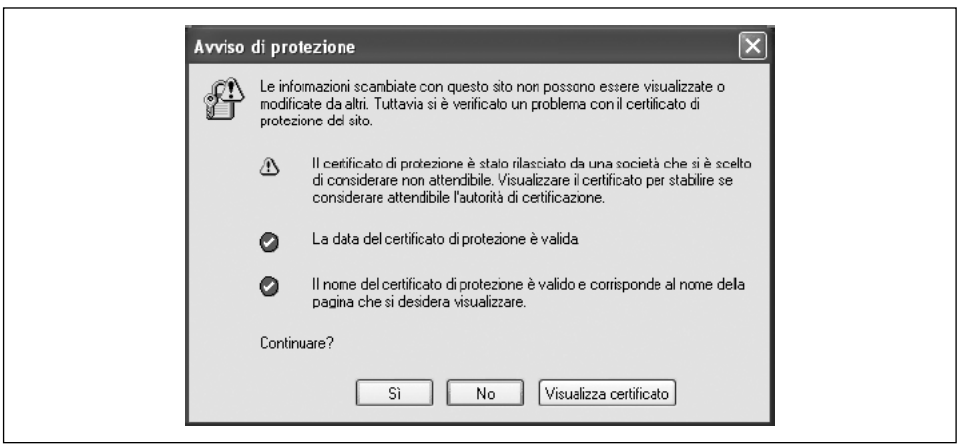

Figura 3.1 Finestra di avvertimento per un certificato non valido in Internet Explorer 6, browser diffuso ancora oggi.

Il testo e l'aspetto degli avvisi SSL si è evoluto negli anni verso spiegazioni incredibilmente stupide (ma sempre problematiche) del problema e azioni sempre più complicate necessarie per superare gli avvisi. Questa tendenza non ha sortito gli effetti sperati: alcune ricerche mostrano che più del 50% degli avvisi più terrorizzanti viene scavalcato dagli utenti[24]. È facile dare la colpa agli utenti, ma – alla fine – si fanno loro le domande sbagliate e si offrono loro esattamente le scelte sbagliate. In tutta franchezza, se è opinione comune che saltare gli avvertimenti sia vantaggioso in alcuni casi – offrire di aprire la pagina in una modalità chiaramente etichettata come "sandbox" dove i danni risulterebbero limitati sarebbe una soluzione decisamente più azzeccata. Quando invece tale convinzione non c'è, si dovrebbe eliminare ogni possibilità di superare il blocco (obiettivo del *Strict Transport Security*, meccanismo sperimentale che vedremo nel Capitolo 16).

Promemoria di ingegneria della sicurezza

Nella gestione dei nomi di file controllati dall'utente negli header Content-Disposition

- ☑ **Se non avete bisogno di caratteri non latini:** tagliate o sostituite ogni carattere che non sia alfanumerico, "." "–" e "_". Per proteggere gli utenti da nomi di file potenzialmente dannosi, potreste verificare che almeno il primo carattere sia alfanumerico e sostituire tutti i punti tranne l'ultimo con qualcos'altro (per esempio un underscore "_"). Tenete presente che permettere virgolette, punti e virgola, backslash e caratteri di controllo (0x00–0x1F) introduce delle vulnerabilità.
- ☑ **Se avete necessità di nomi non latini:** dovete usare i documenti RFC 2047, RFC 2231 o la codifica a percentuale in base al browser in uso. Assicuratevi comunque di filtrare i caratteri di controllo (0x00–0x1F) e di codificare punti e virgola, backslash e virgolette.

Inserimento di dati utente nei cookie HTTP

- ☑ **Codificate a percentuale tutto tranne i caratteri alfanumerici.** O, ancora meglio, usate la codifica base64. Caratteri spuri come virgolette, caratteri di controllo (0x00–0x1F), caratteri a bit alto (0x80–0xFF), virgole, punti e virgola e backslash possono consentire l'iniezione di valori arbitrari o la variazione di significato e ambito d'uso dei cookie.

Invio di header Location controllati dall'utente

- ☑ **Consultate il promemoria del Capitolo 2.** Interpretate e normalizzate gli URL e verificate che lo schema sia nella whitelist di quelli consentiti. Controllate se è il caso di reindirizzare la connessione all'host specificato. Assicuratevi che ogni carattere di controllo e carattere a bit alto sia codificato correttamente. Utilizzate Punycode per i nomi degli host e la codifica a percentuale per il resto dell'URL.

Invio di header Redirect controllati dall'utente

- ☑ **Seguite le indicazioni fornite per Location.** Notate che i punti e virgola sono pericolosi in questo header e non possono essere codificati in sicurezza – oltretutto hanno significati particolari in alcuni URL. Potete rifiutare tali URL del tutto o codificare a percentuale il carattere ";" violando le regole sintattiche dei documenti RFC.

Costruire altri tipi di richieste o risposte controllate dall'utente

- ☑ **Esaminate la sintassi e i potenziali effetti collaterali dell'header in questione.** In generale, dovete avere padronanza di caratteri a bit alto, virgole, virgolette, backslash e punti e virgola; altri caratteri e sequenze possono preoccuparvi caso per caso. Codificate o sostituite tali caratteri quando opportuno.
- ☑ **Nell'implementare un nuovo client, server o proxy HTTP:** non create una nuova implementazione a meno che non dobbiate davvero farlo. Leggete questo capitolo con attenzione e cercate di copiare il comportamento di una implementazione diffusa. Se possibile, ignorate i consigli dei documenti RFC sulla tolleranza agli errori e interrompete connessioni e navigazione se incontrate ambiguità sintattiche.