Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Anno Accademico 2013/2014

Web applications security: HTTP protocol

Srdan Matic <srdjan@security.di.unimi.it>
Aristide Fattori <joystick@security.di.unimi.it>

4 Dicembre 2013

Obiettivi e difficoltà nel raggiungerli

Navigare sicuri all'interno del web

L'utente dovrebbe avere la possibilità di navigare senza incorrere in *pericoli*:

- no sottrazione di informazioni sensibili (se l'utente non intende condividerle)
- no danni/modifiche ai propri file memorizzati sul computer o alla macchina stessa
- visita al sito X non deve interferire con eventuale sessione associata al sito Y

Applicazioni web *sicure*

Vogliamo avere a disposizione delle applicazioni, accessibili tramite web, che siano sicure tanto quanto le normali applicazioni *stand-alone*.

Perchè è così importante?

Alcune statistiche

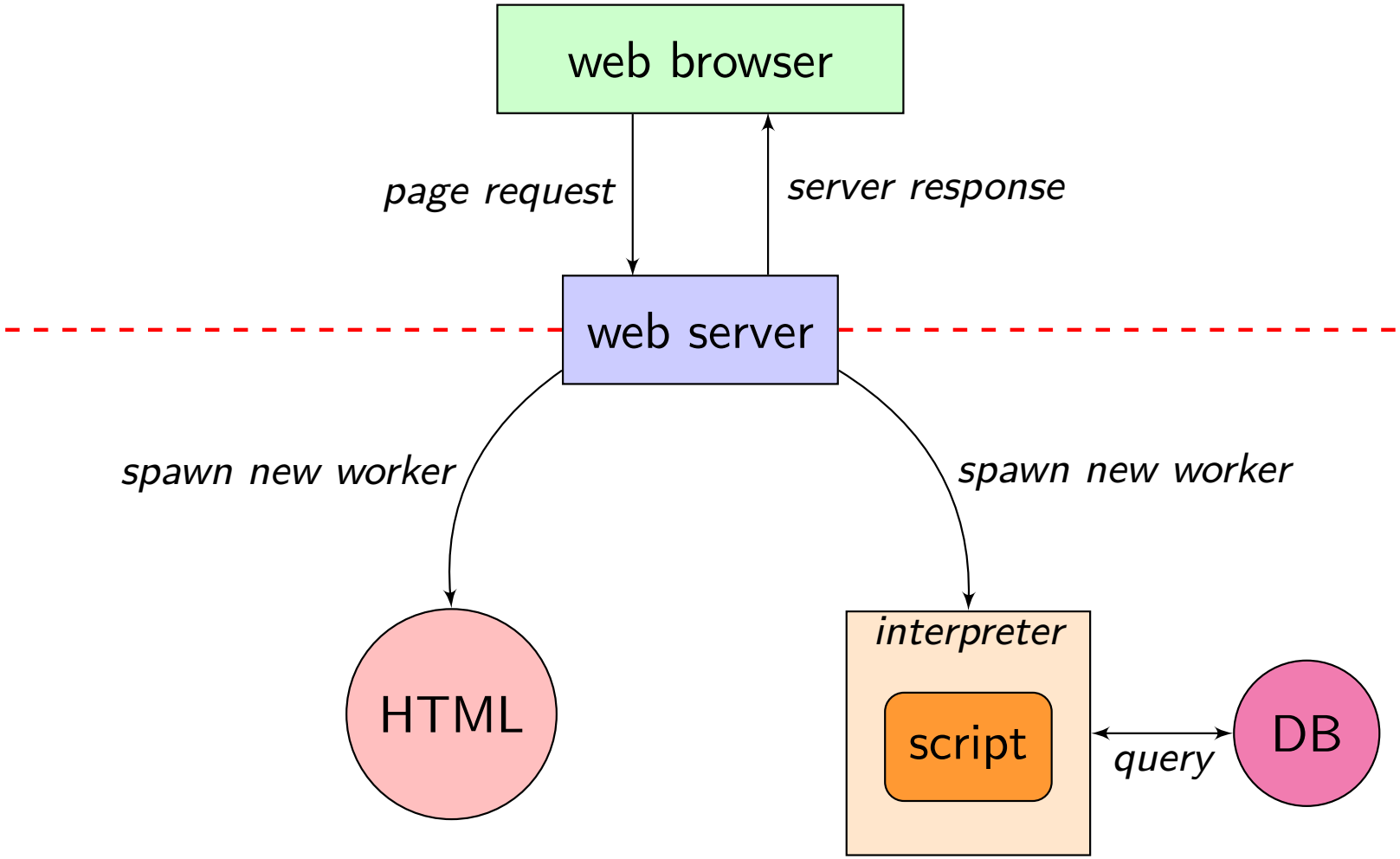
"...The top 10 application vulnerabilities were determined by combining vulnerability risk with frequency of observation. In addition to ranking top vulnerabilities, percentages of applications that contain at least one instance of the vulnerability are also documented."

Top 10 Application Vulnerabilities

RANK*	Finding	Percentage of Applications Containing Vulnerability
1	SQL Injection	15%
2	Miscellaneous Logic Flaws	14%
3	Insecure Direct Object Reference	28%
4	Cross-Site Scripting (XSS)	82%
5	Failure to Restrict URL Access	16%
6	Cross-Site Request Forgery	72%
7	Other Injection	7%
8	Insecure File Uploads	10%
9	Insecure Redirects	24%
10	Various Denial of Service	11%

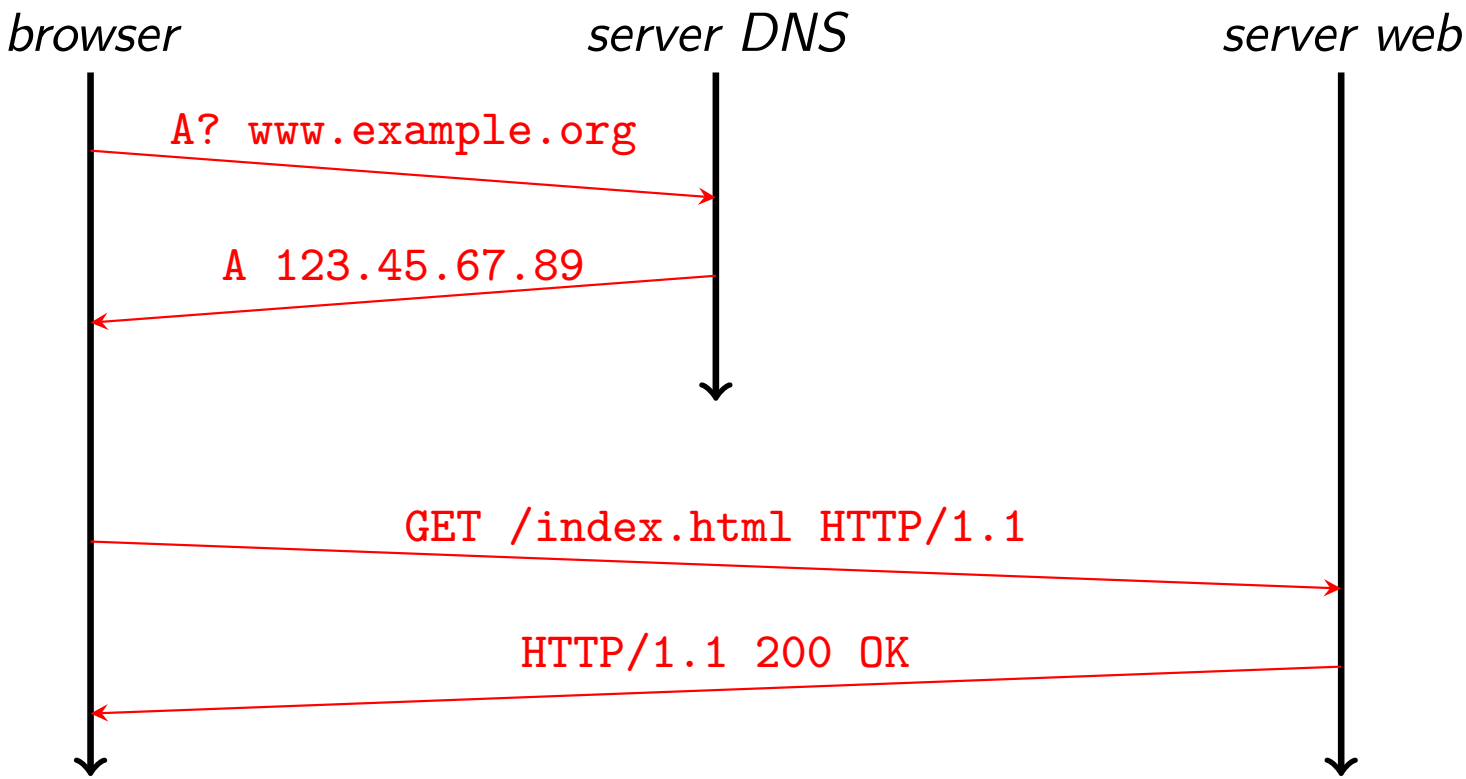
Fonte: `http://www2.trustwave.com/rs/trustwave/images/2013-Global-Security-Report.pdf`

Web - funzionamento



Web - DNS

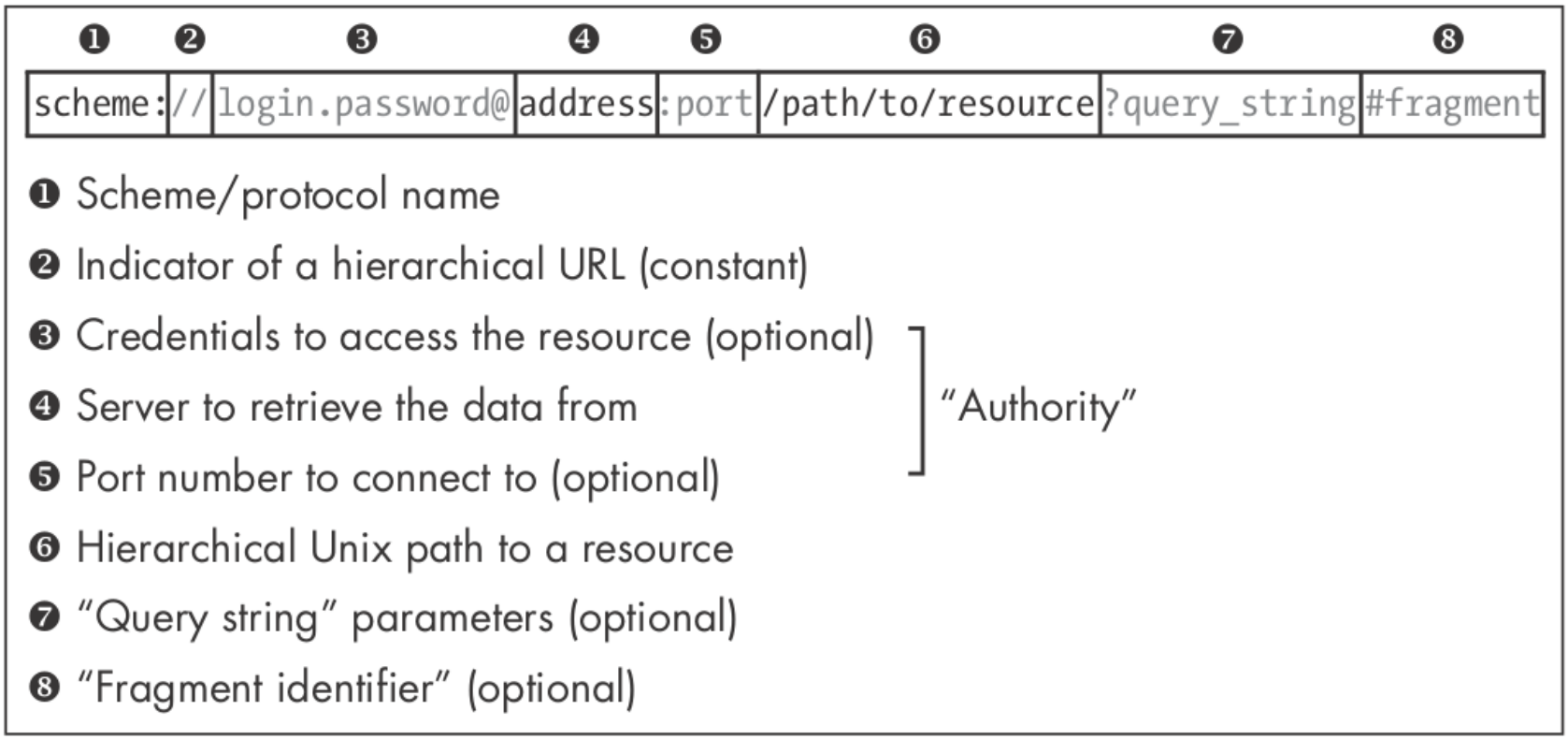
```
srdjan@salvador:$ wget http://www.example.org/index.html
```



client ↔ server DNS

- il *browser* invia una *query* al DNS per ottenere l'indirizzo IP associato ad un particolare domino

Web - struttura del URL



FONTE: "The Tangled Web", di Michael Zalewksi, ED. No Starch Press, 2011.

Componenti del URL

Scheme

- è una stringa *case-insensitive* che termina con il carattere ":"
- i protocolli più comunemente utilizzati sono: "http", "https", "ftp"
- è possibile specificare anche dei *pseudo-URL* ("*javascript:*", "*data:*", "*view-source:*", ...)

Indicator of a hierachical URL

Per soddisfare i requisiti del RFC 1738, si richiede che per ogni URL assoluto e gerarchico la stringa "//" preceda la sezione riservata all'*Authority*.

Componenti del URL

Credentials to access the resource

- come le credendizali vengano inviate al *server* dipende esclusivamente dal protocollo

Server address

È consentito che sia rappresentato tramite:

- un *case-insensitive DNS name*
- indirizzo IPv4
- indirizzo IPv6 racchiuso tra i caratteri “[” e “]”

Server port

Ogni protocollo che si basi su TCP o UDP utilizza numeri di porta di *16 bit* per distinguere messaggi destinati a servizi in esecuzione sulla stessa macchina.
Ogni schema ha una propria porta di *default*.

Componenti del URL

Hierarchical file path

Utilizza la semantica UNIX per indicare una particolare risorsa presente sul *server*.

Query string

La maggior parte degli sviluppatori è solita ad utilizzare *query string* del tipo:

```
name1=value1&name2=value2...
```

Tuttavia questo *layout* non è obbligatorio!

Fragment ID

È stato pensato per fornire informazioni aggiuntive *al client*; solitamente viene utilizzato per memorizzare una *anchor* nel documento HTML e agevolare la navigazione dell'utente.
Può però venire impiegato per altri utilizzi, come ad esempio la memorizzazione di informazioni *sullo stato*.

Caratteri riservati e percent encoding

Lista di caratteri indicati nel RFC 1630 come “non consentiti”.

```
: / ? # [ ] @ ! $ & ’ ( ) * + , ; =
```

Domande:

- perchè sono non consentiti?
- come possiamo fare se abbiamo bisogno di ulizzarli?

Percent encoding

PROTOCOLLO: http **HOST:** gyros **PORTA:** 8080
RISORSA: /l@ser.html **IP:** 192.168.1.8

Esempi di URL codificati

http:gyros/l@ser.html ✗

http://gyros:8080/l%40ser.html ✓

http://%67%79%72%6F%73:8080/l%40ser.html ✓

http://%25%36%37%79%72%6f%73:8080/l%40ser.html ✗

http://gyros:%3A%38%30%38%30/l%40ser.html ✗

http://192.168.1.%38:8080/l%40ser.html ✓

http://www.ręczniki.pl/ręcznik?model=Jaś#Złóż_zamówien ✗

Codifica *Punycode*

Esempio

Seppure visivamente identiche, le due stringhe identificano due *host*/macchine differenti[a]:
`http://example.org/` ← u“http://\u0453xample.org/”
`http://example.org/` ← u“http://example.org/”

[a] http://jrgraphix.net/r/Unicode/0400-04FF

I nomi di domino sono sempre codificati in caratteri ASCII.
In seguito all'introduzione della possibilità di registrare domini con caratteri *Unicode*, è stata ideata la codifica nota come *Punycode*.
Il *Punycode*, definito nel RFC 3492, consente di mappare univocamente una stringa *Unicode* nel suo equivalente ASCII.
È compito del *browser* effettuare la conversione da *Unicode* a *Punycode*.

`http://www.ręczniki.pl/`
⇓
`http://www.xn--rczniki-98a.pl/`

Il protocollo HTTP

HTTP: HyperText Transfer Protocol

HTTP:

- originariamente pensato per il trasferimento di documenti in formato HTML, oggigiorno è alla *base* del Web
- è un protocollo di *livello applicazione* che viene utilizzato per trasferire dati tra un *client* ed un *web server*
- si basa sullo scambio di semplici messaggi di testo
- attualmente vengono utilizzate due versioni: 1.0 (RFC 1945) e 1.1 (RFC 2616)
- dati incapsulati su una *connessione TCP* (*default*: porta 80)
- è *stateless*

Il protocollo HTTP

Comunicazione *client* $\leftrightarrow$ *server* tramite HTTP:

- Il *client* crea una connessione TCP con il *server* ed effettua la richiesta di una particolare risorsa
- il *server* invia al *client* la risorsa richiesta, quindi chiude la connessione TCP al termine dell'invio

Evoluzione del protocollo

- *draft* di HTTP /0.9 sviluppato da Tim Berners-Lee nel 1991 ed era lungo poco più di una pagina
- nel 1995 viene rilasciato RFC 1945, che definisce HTTP/1.0; lunghezza $\approx$ 50 pagine
- RFC 2616 definisce lo standard per HTTP/1.1 e viene rilasciato nel 1999; la sua dimensione supera le 150 pagine

Oggi la maggior parte applicazioni utilizzate lato *client* e *server* supporta "gran parte delle specifiche" definite in HTTP/1.0 e HTTP/1.1

Funzionamento di HTTP

Esempio di richiesta utilizzando HTTP/0.9

GET /TimBern.htmlCRLF[a]

[a] CR == "\x0D" e LF == "\x0A"

In risposta alla precedente risposta il server restituisce esclusivamente il *payload* del rispettivo documento HTML.

È un protocollo robusto ed efficiente?

- come specifichiamo eventuali preferenze linguistiche dell'utente?
- come indichiamo i *tipi* di documenti che vogliamo ricevere in risposta?
- come capiamo se il documento effettivamente è presente sul *server*? o se dobbiamo invece cercarlo altrove?
- se indichiamo solo *path+query string*, come possiamo ospitare differenti siti web su un unico server? come facciamo a capire a quale sito intendiamo inoltrare la richiesta?

HTTP request

HTTP/1.0 e HTTP/1.1 introducono modifiche alle richieste dei client imponendo l'utilizzo del seguente formato:

Struttura:

1. first line
   - `METHOD PATH+QUERY_STRING PROTOCOL_VERSION`
2. headers (*opzionali*)
3. empty line
4. payload (*opzionale*)

Note

- al termine di "first line" e "headers" *dovrebbe* comparire la sequenza di caratteri `CRLF`.
- tutte le righe che compongo gli "headers" sono nel formato "name: value"
- empty line $\rightarrow$ `CRLF`

HTTP response

HTTP/1.0 e HTTP/1.1 introducono modifiche alle risposte dei server imponendo l'utilizzo del seguente formato:

Struttura:

1. first line
   - `PROTOCOL_VERSION STATUS_CODE TEXT_MESSAGE`
2. headers (*opzionali*)
3. empty line
4. payload (*opzionale*)

Note

- al termine di "first line" e "headers" *dovrebbe* comparire la sequenza di caratteri `CRLF`.
- il `TEXT_MESSAGE` al termine di "first line" è opzionale
- tutte le righe che compongo gli "headers" sono nel formato "name: value"
- empty line $\rightarrow$ `CRLF`

Esempio di comunicazione client-server tramite HTTP/1.1

HTTP request

```
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 <-- !

I REQUEST A BUNNY
```

HTTP response

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

BUNNY WISH HAS BEEN GRANTED
```

Anche in assenza del header "Content-Lenght" l'output visualizzato da Chrome e Firefox è analogo
... ma se invece provassimo a utilizzare HTTP/1.0 cambierebbe qualcosa?[a]

[a] https://bugzilla.mozilla.org/show_bug.cgi?id=475444#c6

Cosa sono e perchè vengono impiegati

I *proxy* vengono utilizzati spesso per intercettare, ispezionare e inoltrare richieste HTTP.
L'Utilizzo di *proxy* avviene per diversi motivi: migliorare le *performance*, imporre l'utilizzo di particolari *policy*, controllare e monitorare l'accesso a specifici servizi che risiedono in reti differenti.

Tipologie di proxy

I *proxy* possono venire classificati secondo due tipologie:

- *transparent proxy*
- *non-transparent proxy*

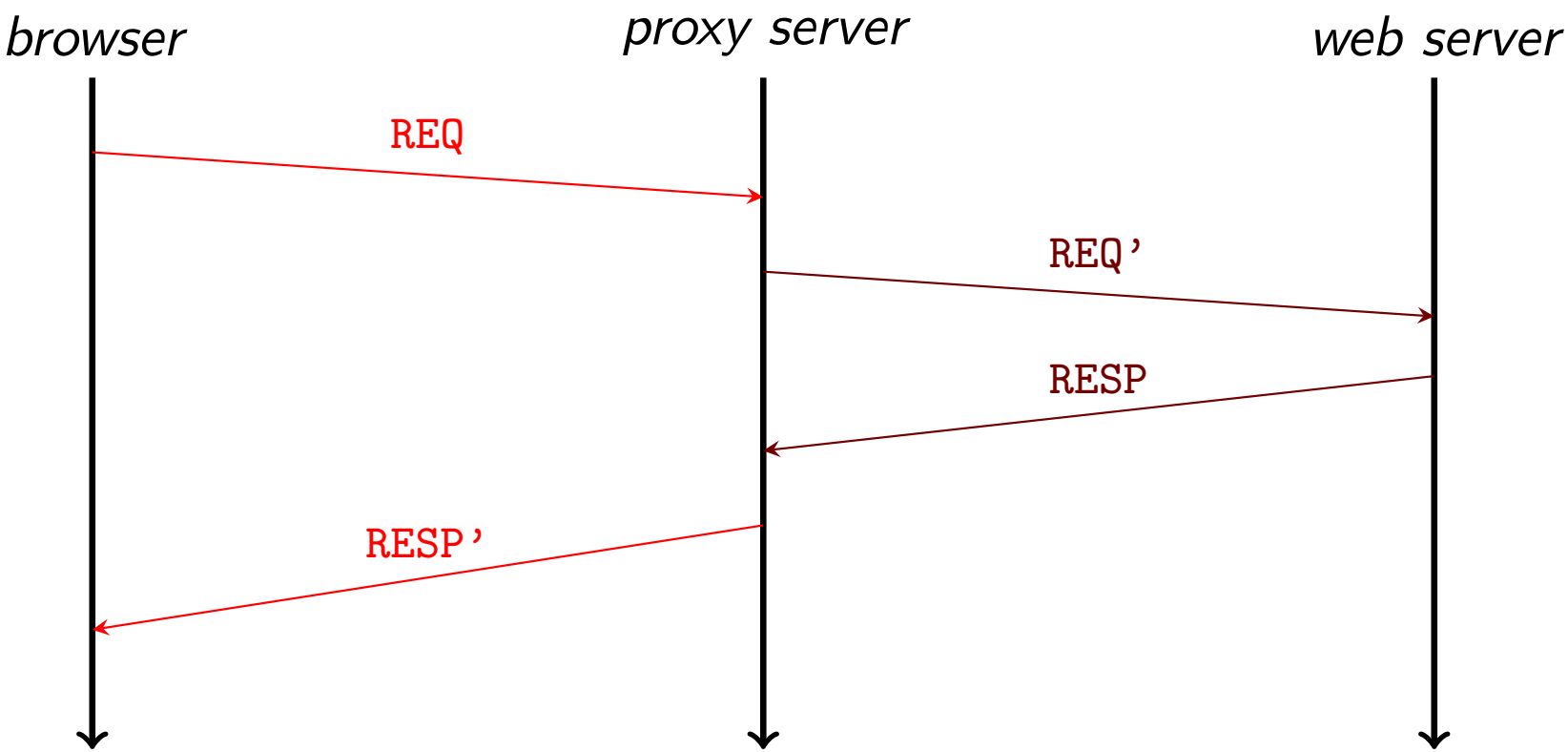
Add-on ed extension per i browser

Chrome/Chromium: `Proxy SwitchySharp` - `Proxy Switchy!`
Firefox/Iceweasel: `FoxyProxy` - `QuickProxy` - `Hide My Ass! Web Proxy`

Richiesta tramite proxy HTTP

Esempio di richiesta tramite proxy HTTP

```
GET http://www.ccdinf.unimi.it/it/index.html HTTP/1.1
Host: www.ccdinf.unimi.it
User-Agent: Mozilla/4.0 (compatible; MSIE 6.01; Windows NT 6.0)
...
```



Il header Referer

Il *header* "Referer" viene utilizzato per indicare l'URL che ha generato la richiesta ad una specifica risorsa.

Problemi?

- potrebbe rivelare informazioni riguardo alla navigazione dell'utente
- potrebbe contenere informazioni sensibili nei "query parameters" del URL (ex. Siamo arrivati a `http://www.google.com/` passando da `http://www.example.org/private_admin_section.php?action=logout&user_id=0`)

Situazioni in cui non viene incluso tra gli altri *header*

- inserimento manuale dell'URL nella barrra o visita selezionando il *link* dai *bookmark*
- se arriviamo al URL destinazione passando da uno *pseudo-url*
- quando l'URL di partenza è all'intero di una sessione cifrata, ma quello destinazione *non* lo è
- se viene bloccato dall'utente tramite l'installazione di appositi *plugin* e *add-on*

Tipi di HTTP request

GET

- è il metodo più utilizzato per le comuni interanzioni *client-server*
- *non* è prevista la presenza di *payload* nella *request*
- RFC suggeriva che "*GET requests should not have other significance of taking an action other than retrieval*"

POST

- è stato pensato per l'invio di informazioni (tipicamente sotto forma di *form* HTML)
  - queste azioni hanno *modifiche persistenti* sullo stato dell'applicazione; per questo motivo il *browser* chiede conferma prima di replicare una richiesta POST
- quando si utilizza POST, è sempre presente il header "Content-Length"

Tipi di HTTP request

HEAD

- è analogo a GET, ma il *server* restituisce in *output* solo gli *header*
- tipicamente utilizzato da *search engine bots* e altri *tool* automatici

OPTIONS

- specifica quali metodi è possibile utilizzare per richiedere una particolare risorsa
- mettendo * al posto del URL, restituisce i metodi supportati dal *server*

PUT e DELETE

- originariamente pensati per consentire invio e rimozione di file sul *server*
- nella pratica non vengono mai impiegati (sostiuiti da POST/GET + *server side script*)

Tipi di HTTP request

TRACE

- è una sorta di *ping*
- restituisce utili informazioni sui *proxy* intermedi presenti tra *client* e *server*
- in *output* al *client* viene restituita l'intera richiesta originale

Tipi di HTTP response

RFC 2616 definisce $\approx$ 50 differenti *status code* tra cui il *server* può scegliere per costruire la *response* che verranno inviate al *client*. Nella pratica meno di 1/3 di essi viene effettivamente impiegato.

200-299 $\rightarrow$ Success

Codici all'interno di questo intervallo indicano che la *request* è andata a buon fine.

- ***200 OK***: risposta normale a fronte di una GET o POST, che ha avuto esito positivo. Il contenuto del *payload* viene visualizzato all'utente.
- ***204 No Content***: la richiesta è andata a buon fine, ma non vi nessun è contenuto da visualizzare.
- ***206 Partial Content***: viene utilizzato in combinazione con le "*range request*"; i dati vengono ri-assemblati utilizzando le informazioni contenute nel apposito *header* "Content-Range".

Tipi di HTTP response

300-399 → Redirect

Codici all'interno di questo intervallo indicano che non si è verificato un errore, ma il *browser* deve effettuare un'operazione specifica.

- ***301 Moved Permanently***, ***302 Found***, ***303 See Other***: si richiede al *browser* di inoltrare la richiesta altrove, utilizzando le informazioni contenute nel *header* "Location"
- ***304 Not Modified***: la risorsa non è cambiata rispetto alla versione di cui il *client* è già in possesso (tipicamente si può ottenere in presenza di *header* come *If-Modified-Since*).
- ***307 Temporary Redirect***: analogo a ***302***, ma la richesta non viene inviata utilizzando esclusivamente il metodo GET.

Tipi di HTTP response

Codici all'interno di questo intervallo indicano errori legati al comportamento del *client*.

400-499 → Client-Side Error

- ***400 Bad Request***: il *server* non è in grado di (o non intende) processare quella specifica richiesta; informazioni più dettagliate vengono solitamente incluse all'interno del *payload*.
- ***401 Unauthorized***: è richiesto che l'utente fornisca le proprie credenziali prima di richiedere la risorsa.
- ***403 Forbidden***: la risorsa esiste ma non è possibile accedervi (la causa tipicamente sono problemi di configurazione e/o permessi a livello del *filesystem*).
- ***404 Not Found***: l'URL richiesto non esiste.

500-599 → Server-Side Error

Codici in questo intervallo indicano errori legati a problemi (principalmente di configurazione) verificatisi lato *server*.

Passaggio di parametri - metodo GET

- l'utente invia dati all'applicazione web tramite una *form* o una qualsiasi altra tecnologia *client-side* (esempio JavaScript)
- le informazioni immesse devono essere trasformate in una *HTTP request*

Scenario 1: passaggio di parametri tramite *form*

```
<form action="submit.php" method="get">
   <input type="text" name="var1" />
   <input type="hidden" name="var2" value="b"/>
   <input type="submit" value="send" />
</form>
```

Scenario 2: parametri già contenuti all'interno del URL

```
<a href="submit.php?var1=a&var2=b">link</a>
```

Corresponding HTTP request

```
GET /submit.php?var1=a&var2=b HTTP/1.1
Host: www.example.com
...
```

Passaggio di parametri - metodo POST

Scenario 1: *form* che utilizza il metodo POST

```
<form action="submit.php" method="post">
   <input type="text" name="var1" />
   <input type="hidden" name="var2" value="b"/>
   <input type="submit" value="send" />
</form>
```

```
POST /submit.php HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

var1=a&var2=b
```

Scenario 2: GET + POST

È consentito?

```
<form action="submit.php?var3=c&var4=d"
      method="post">
   <input type="text" name="var1" />
   <input type="hidden" name="var2"
          value="2 + 3"/>
   <input type="submit" value="send" />
</form>
```

```
POST /test.php?var3=c&var4=d HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

2+%2B+3
```

HTTP Authentication

I meccanismo di autenticazione introdotto nel RFC 2616, oggigiorno viene utilizzato molto raramente.
Funzionamento:

1. *browser* effettua una richiesta senza le credenziali del *client*
2. il *server* risponde con uno *status message* "*401 Unauthorized*" (includendo un apposito header *WWW-Authenticate* che contiene informazioni sul metodo di autenticazione).
3. il *browser* ottiene le credenziali dal client e le include nell'apposito header *Authorization*

Per inviare le credenziali al *server* è possibile scegliere tra

- ***basic***: la *password* inviata viene codificata in *base64*
- ***digest***: le credenziali vengono inviate al *server* dopo avere calcolato una *hash* sulla *password* combinata con con un *token* generato dal *server*.

Analisi e manipolazione di traffico HTTP

Traffico HTTP

- *header* e *payload* vengono incapsulati all'intero di pacchetti TCP (default: porta 80)
- tutte le comunicazioni avvengono con il testo in in chiaro

$\Longrightarrow$ Monitorare && analizzare il traffico è molto semplice!!!

Perchè dovremmo analizzarlo?

- analisi *black-box* di applicazioni web

Come facciamo?

- avete a disposizione un arsenale di *tool* per fare *sniffing* di traffico (e.g.,`ngrep`, `tcpdump`, `wireshark`, ...)

HTTP manipulation

- *extension* e *addon* del *browser*
- proxy
- **netcat**, `curl`, ...

HTTPS manipulation

- apposite *browser extension* (e.g., Firefox $\rightarrow$ `Tamper Data`)
- proxy

Proxy HTTP

- vengono impiegati per analisi ed eventuale modifica del traffico`HTTP`/`HTTPS`
- sono indipendenti dall'applicazione
- se utilizziamo un proxy per intercettare traffico HTTP, il *browser* ci sengalerà un errore nella fase di verifica del certificato SSL...

Alcuni proxy HTTP

- `WebScarab` (`http://www.owasp.org/`)
- `ProxPy` `http://code.google.com/p/proxpy/`
- **Burp** (`http://www.portswigger.net/proxy/`)

DEMO

http://gamebox.laser.dico.unimi.it/aa1314_sec2_web1

Content Isolation Logic

Content Isolation

Gran parte dei meccanismi di sicurezza forniti dal *browser* fanno affidamento sulla possibilità di isolare documenti (e contesti di esecuzione) in base all'origina della risorsa. L'idea è infatti molto semplice:

> "The pages from different sources should not be allowed to interfere with each other".

Riuscire a identificare tutti i punti in cui occorre effetuare *security check* non è banale:

- azioni come "seguire un link", dovrebbero essere sempre consentite ✗
- azioni del tipo "modificare il contenuto di una applicazione/pagina" dovrebbero sempre venire verificate prima di venire eseguite ✗

SOP - *Same Origin Policy*

SOP viene introdotto nel 1995 da Netscape, 1 anno dopo la standardizzazione del meccanismo dei *cookie*.

Prerequisiti per SOP

Dati 2 contesti di esecuzione di due script, essi possono accedere ai rispettivi DOM se e solo se il protocollo, il DNS name e la porta da cui provengono i due script, sono uguali.

Originating document	Accessed document	Non–IE browser	Internet Explorer
http://example.com/**a/**	http://example.com/**b/**	Access okay	Access okay
http://example.com/	http://**www.**example.com/	Host mismatch	Host mismatch
http://example.com/	**https**://example.com/	Protocol mismatch	Protocol mismatch
http://example.com**:81**/	http://example.com/	Port mismatch	Access okay

FONTE: "The Tangled Web", di Michael Zalewksi, ED. No Starch Press, 2011.

SOP - Limiti e soluzioni

Semplicità di SOP è al tempo stesso un suo limite

- non possiamo isolare *homepage* che appartengono a utenti differenti ma che provengono dallo stesso sito/*host*
- non possiamo fare cooperare siti che avrebbero bisogno di interagire (es. `http://store.google.com` e `http://play.google.com`)

Soluzioni:

1. `document.domain`: entrambi gli *script* possono impostare come dominio di controllo per SOP il loro comune *top level domain* (es. `http://google.com`)
   - questo però può portare a nuovi problemi (es. $\rightarrow$ i precedenti due *script* ora saranno in grado di comunicare anche con `http://mobile.google.com`)
2. postMessage(...): versione più sicura, API introdotta con HTML5

HTTP sessioni & *cookie*

HTTP Sessioni - Introduzione

Problemi

- **stateless**: ogni richiesta è indipendente dalla precedenti
- le applicazioni web hanno la necessità di avere un meccanismo per la tenere traccia delle sessioni

Per cosa vengono utilizzate le sessioni?

- evitare di dovere fare *login* su ogni singola pagina
- tenere traccia delle preferenze dell'utente
- tenere traccia delle azioni effettuate dall'utente (ad esempio saper quali oggetti sono stati aggiunti al carrello virtuale, ...)
- ...

HTTP Sessioni - Possibili soluzioni

- le sessioni vengono implementate all'interno delle applicazioni web grazie all'utilizzo di appositi *script*
- le informazioni riguardanti la sessione fluiscono tra il *client* e il *web server*

Come trasmettiamo le informazioni che riguardanti le sessioni?

1. payload HTTP

```
<INPUT TYPE=hidden NAME=sessionid VALUE=7456»
```

2. URL

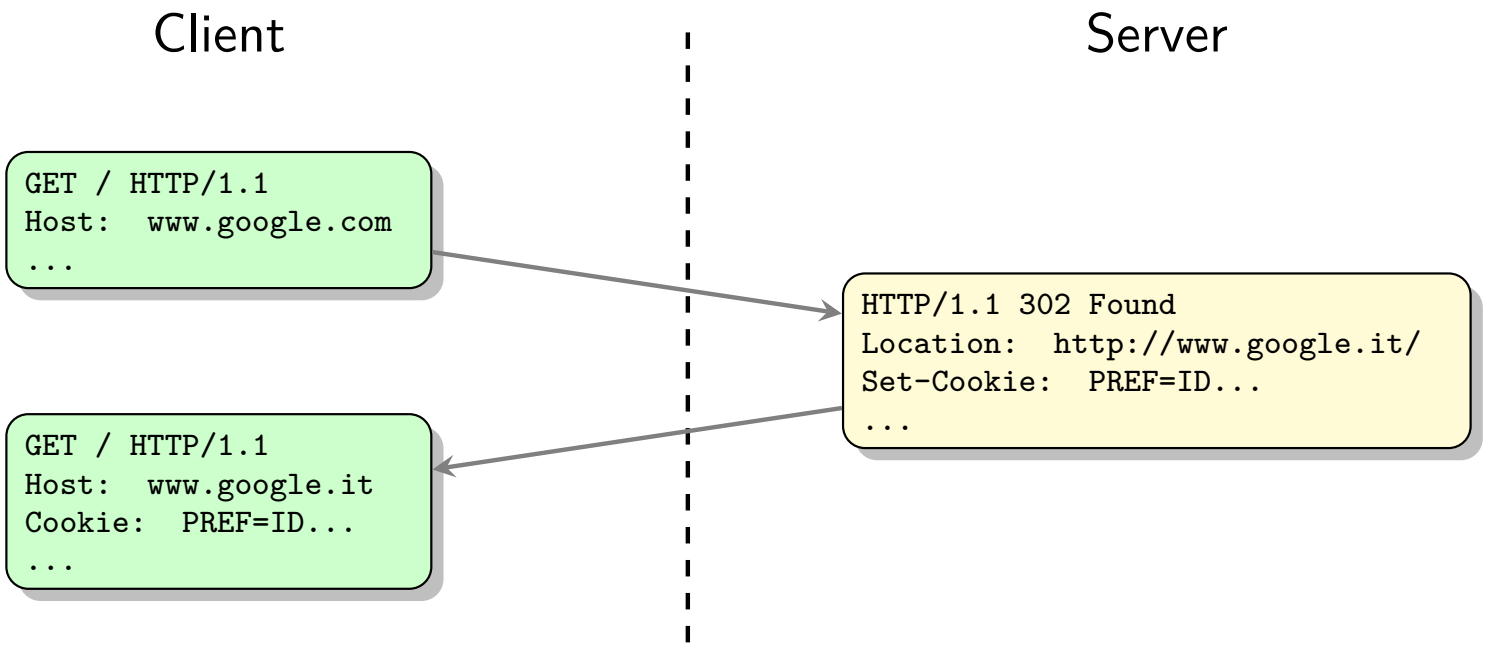
```
http://www.example.com/page.php?sessionid=7456
```

3. header HTTP (e.g., Cookie)

```
GET /page.php HTTP/1.1
Host: www.example.com
Cookie: sessionid=7456
...
```

Funzionamento

- il *cookie* rappresenta un identificatore univoco che il *server* associa al *client* per potere tenere traccia della sessione
- l'informazione viene generata lato *server* quindi inviata al *client*
- in tutte le comunicazioni successive il *client* invierà al *server* il *cookie* per consetire l'idenficazione della sessione a cui fare riferimento



Cookie - attributi

Informazioni dettagliate sui *cookie*, il loro funzionamento e gli attributi che possono venire utilizzati, si possono trovare nel apposito RFC 2109.
Il formato standard per assegnare un valore ad un *cookie* è:

```
name=value
```

Attributo	**Funzione**
`Expires`	indica fino a quando il *cookie* deve essere considerato valido
`Max-age`	simile al precedente, ma raramente utilizzato
`Domain`	il *cookie* può essere impostato come valido per un domino più generico da quello da cui proviene
`Path`	il *cookie* può venire associato ad uno specifico *path*
`Secure`	il *cookie* deve venire inviato solo su connessioni sicure
`HttpOnly`	specifica che non è consentito utilizzare la API `documento.cookie` di JavaScript, per accedere al *cookie*

Cookie && SOP

I *cookie* sono stati ideati per essere associati a domini, di conseguenza non è possibile associarli ad uno specifico *hostname*.
Ad esempio, *cookie* provenienti da `http://foo.example.org` potrebbero venire inviati sia a `http://bar.foo.example.org` che a `http://example.org` !
In *alcuni browser* è possibile associare i *cookie* ad uno specifico dominio, semplicemente omettendo l'attributo *Domain*.

Cookie set at *foo.example.com*, *domain* parameter is:	**Scope of the resulting cookie**	
	Non–IE browsers	**Internet Explorer**
(value omitted)	*foo.example.com* (exact)	**.foo.example.com*
bar.foo.example.com	Cookie not set: domain more specific than origin	
foo.example.com	**.foo.example.com*	
baz.example.com	Cookie not set: domain mismatch	
example.com	**.example.com*	
ample.com	Cookie not set: domain mismatch	
.com	Cookie not set: domain too broad, security risk	

Sessioni tramite cookie

Per realizzare una sessione abbiamo a disposizione due opzioni:

1. le informazioni riguardanti la sessione vengono inserite nell'applicazione web dallo sviluppatore stesso (tecnica obsoleta e poco sicura)
2. la sessione viene implementata utilizzando il linguaggio di programmazione con cui è realizzata l'applicazione web stessa

Cookie e sessioni

- è l'approccio più comunemente adottato
- le informazioni della sessione vengono memorizzate **sul server**
- il *server* invia un ID al *client* utilizzando il meccanismo dei *cookie*
- per ogni richiesta successiva, il *client* allega nelle proprie richieste il *cookie* ricevuto dal *server*
  (e.g., `Cookie: PHPSESSID=da1dd139f08c50b4b1825f3b5da2b6fe`)
- il server utilizza le informazioni memorizzate ed associate allo specifico ID, per effettuare ulteriori operazioni

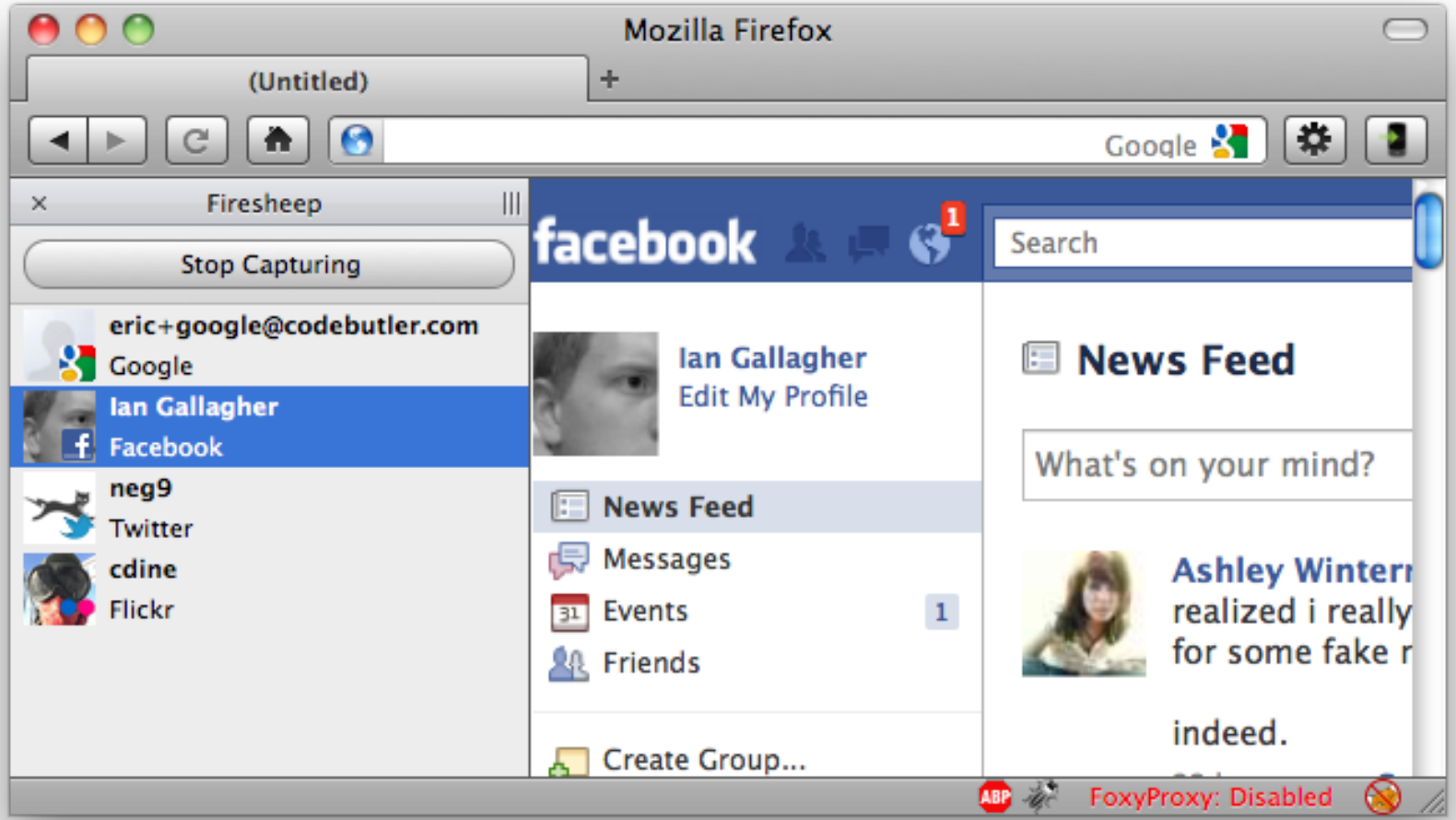
Il meccanismo precedente possiede problemi ed elementi estremamente interessanti dal punto di vista della sicurezza

- il *cookie* diventa un elemento **critico** (in quanto usato per autenticarsi)
- rischio: *bypassare* eventuali schemi di autenticazione!
- i *cookie* di sessione *devono* essere conservati sul server per il **più breve tempo possibile**

Attacco	**Contromisura**
hijacking	`SSL/TLS`
prediction	usare un valido PRNG
session fixation	controllo del `IP`, header `Referer`
brute force	increase `ID` length
stealing (XSS)	prossima lezione

Esempio - Session Hijacking

Esempio - Session Hijacking



Estremamente semplice da realizzare!

(source: `http://codebutler.com/firesheep`)

Esempio - Session Prediction

Slides:

`http://goo.gl/EA06K`

Video:

`http://www.youtube.com/watch?v=XQcW1zYiqdU` - part 1

`http://www.youtube.com/watch?v=Mi1Z2iJZPs4` - part 2

`http://www.youtube.com/watch?v=jMWEZn1_O3Y` - part 3

`http://www.youtube.com/watch?v=kS4MFq3QDS4` - part 4

Paper:

`http://goo.gl/Lkalf`

È più semplie di quanto si potrebbe pensare!

160 bit $\rightarrow \approx$ 20 bit

(source: Black Hat 2010)

Esempio - Session Fixation