

Web Security

Session Management

Recall from last week: the web

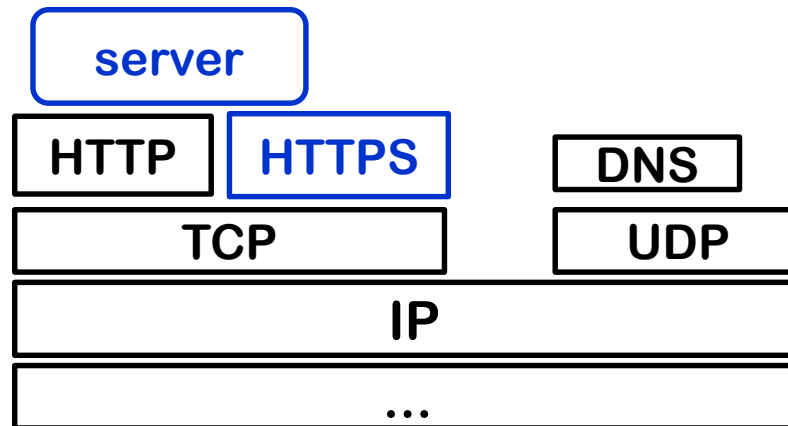
- On the web, servers and clients communicate by **HTTP requests** and **responses**
- HTTP request are usually **GET** or **POST** requests
 - GET: parameters in URL
 - POST: parameters in HTTP body
- Data sent in HTTP traffic, in the URL or in the body, may be encoded in various ways
 - **URL encoding** for (data used in) URLs
 - **HTML encoding** for (data used in) HTML
 - **base 64 encoding** for raw binary data sent as text

Web Security

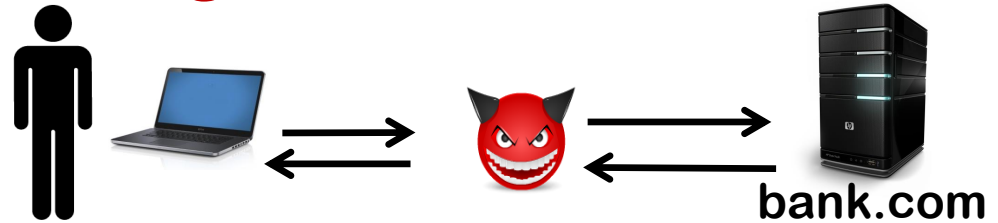
Session Management

Today: two notions of sessions

1. **HTTPS** at the **network layer**, ie using TLS
 - incl. authentication of the server
2. **Session management** at the **application layer**
 - by web-application using sessions IDs and/or cookies
 - incl. authentication of the user



Security shortcoming of internet & web



Attacker model:

passive eavesdropper or *active Man-in-the-Middle (MitM)*
eg malicious/compromised **ISP, router, or WiFi access point**

- No *confidentiality*

All traffic is public and can be **observed**

Not by attacker that just sniffs the WiFi, if we ignore the security flaws of WEP/WPA1/WPA2

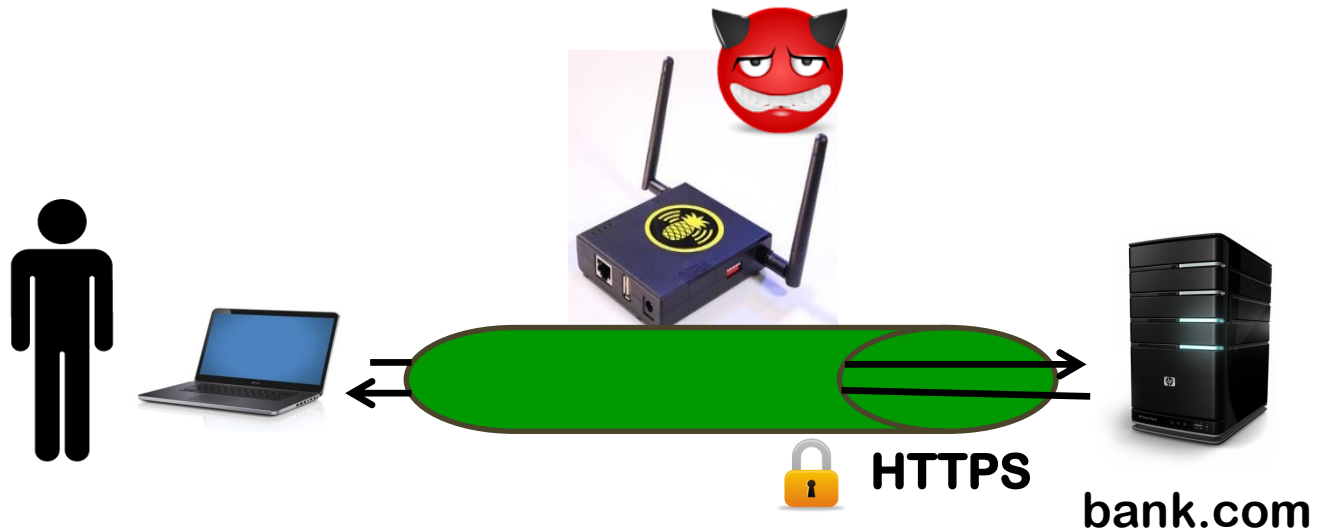
- No *integrity*

All traffic can be **altered** by these intermediate parties

No *authentication*

browser and server don't really know who they are talking to, apart from an IP address

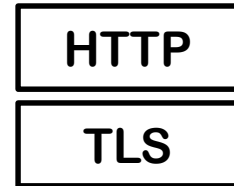
(partial) security solution: TLS



(Aside: name confusion TLS vs SSL)

- **TLS (Transport Layer Security)**
used to be called **SSL (Secure Sockets Layer)**
 - TLS version 1.0 is SSL version 3.1
 - TLS version 1.2 is current, TLS version 1.3 is being rolled out
- In practical use, SSL and TLS are synonyms.
Eg. X509 certificates are often called **SSL certificates**
and a leading TLS implementation is called **OpenSSL**

HTTPS: HTTP over TLS



TLS is configurable, and security guarantees depend on configuration:

- **Confidentiality & integrity of the session**
 - Attacker on the network can still see *that* two IP addresses communicate (i.e. **meta-data**), but not *what*
 - All HTTP content, incl. headers & URL parameters are protected inside TLS tunnel
 - Attacker cannot change any traffic or replay without this being detected
- Nearly always: **server authentication**, using **certificates**
- Possibly, but uncommon: **client authentication** with a client certificate
 - Usually servers use some other means to authenticate clients:
often **passwords**

HTTPS

1. Server sends **X509 server certificate** to client, which
 - includes server's **public key** PK
 - is digitally signed by **Certificate Authority (CA)**, or **self-signed**
 - Browsers come pre-configured with a list of trusted CAs
2. Client checks that certificate has not been revoked
 - by requesting **Certificate Revocation List (CRL)** from CA
3. Client authenticates the server, with a challenge-response protocol
 - Client sends nonce n encrypted with public key PK , and checks if the response includes n , which proves knowledge of the private key
4. Client and server then agree a **session key**
 - typically an AES key, based on n and a random chosen by server
5. Subsequent HTTP traffic in **a secure tunnel**
 - Traffic **encrypted** and **MACed** with session key
 - encryption for **confidentiality**, MACing for **integrity**
 - Periodically the session key is refreshed



Functional shortcoming of HTTP

HTTP is **stateless** and has **no notion of session**, ie

- No state is recorded about previous requests
- (Hence) no notion of a sequence of requests belonging together in one conversation between client and server
 - eg imagine your WhatsApp conversations were not grouped by user, and just showed telephone numbers and not names, and people sometimes changed telephone numbers

This is very clumsy if for interaction between a client and server

- *Has this user logged in?*
- *Has the user put items in online shopping basket?*
- *Did this use already agree to our privacy policy?*

Why can't we use IP address for this?

- **Different clients may share the same IP address**
Eg different users on lilo.science.ru.nl,
or different users on a local wifi network
- **Multiple web applications can share the same IP address**
 - especially web applications hosted in the cloud
- **Clients and servers can change IP address**
 - esp. clients on mobile devices, when switch from mobile network to WiFi or back
 - also web applications hosted in the cloud, if they are migrated to other machines

Session & session data

There is usually **session data** associated with a session that needs to be remembered.

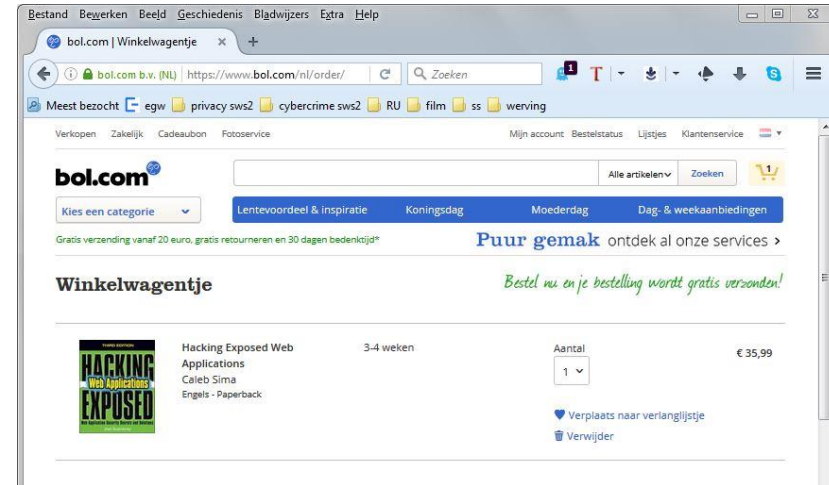
- Eg: content of online shopping basket

Ways of keeping track of such data:

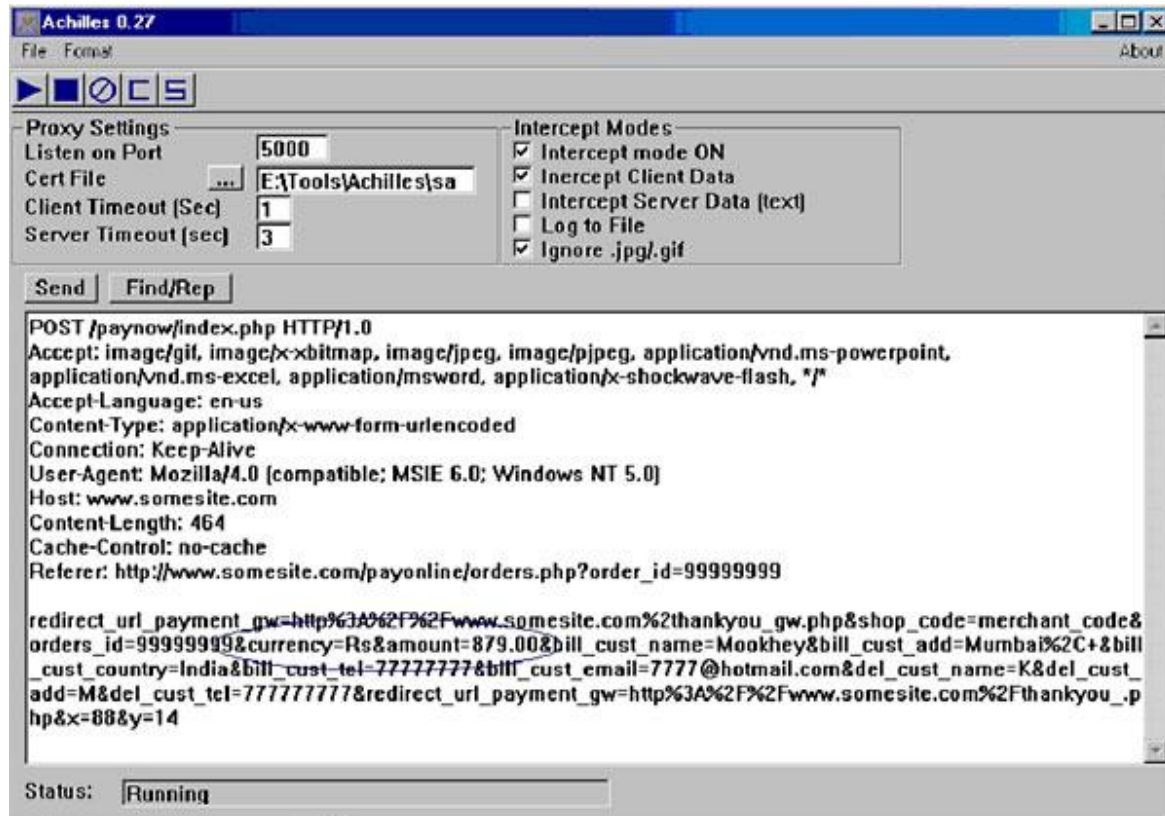
1. **send it back & forth between server and browser**
with each request and response
eg using hidden parameters
2. **record it at the client side**
using HTML5 local storage
3. **record it at the server side** and just send back & forth a unique identifier

Pros & Cons ?

- Con 3: server has to record lots of info for many sessions
- Con of 1 & 2: client could mess with this data



Things that can go wrong with session data



Classic security flaw: the price is recorded in a hidden form field, as shown in the proxy output above.

The client can change this...

Misplaced trust in the client

For data for which integrity is important (eg prices)

the server should never trust the client

to provide this data or to return this data unaltered

Instead, the server should

- store such data server-side,

or

- add a cryptographic integrity check
 - eg using a MAC (Message Authentication Code) or Digital Signature
 - Such a check should also include a **time stamp** or some **session id** that frequently changes, to avoid replay or roll-back attacks.

Sessions managed by the web application

Typical steps

1. Web application creates & manages sessions
 - **Session data** is stored at server and associated with a unique **session ID**
2. Client is informed of session ID
 - and client attaches session ID to subsequent requests so server knows about previous requests

Web application frameworks usually provide built-in support for session management, but web application developers can implement their own

- NB it is better to use existing solutions than inventing your own
- Still, don't underestimate the complexity of using these correctly

Sessions & authentication

The notion of session close tied with authentication

- Eg after logging in with a username & password, you will often have certain access rights for the rest of the session

While the session lasts, *any information that can be used by an attacker to spoof the session (eg a session ID) is just as valuable as the username/password!*

So such info should not be sent via HTTP but via HTTPS.

How can a website mitigate this risk?

- *by having a time-out to terminate inactive sessions*
- *by having a prominent LogOut button on every webpage*

Solution 1: session ID in URL

Web page returned by the server contains links with **session ID as extra parameter**

```
<html>
```

Example web page with session IDs in the URL.

The user can now click

```
<a href="http://demo.net/nextpage.php?sid=1234">here</a>
```

or

```
<a href="http://demo.net/anotherpage.php?sid=1234">here</a>
```

passing on its session id back to the server

wherever he goes next.

```
</html>
```

Hence: every user gets their own unique copy of a web page.

Solution 2: session ID in hidden parameter

<htm>

The form below uses a hidden field

```
<form method="POST" action= "http://ru.nl/register.php">  
  Email: <input type="text" name="Your email address">  
  <input type="hidden" name="sid" value="s1234">  
  <input type="submit" value="Click here to submit">  
</form>
```

Hidden means hidden by default by browser, *not* hidden from a proxy like ZAP.

A hidden form field could also be used to track user preferences, eg

```
<input type="hidden" name="Language" value="Dutch">
```

Session ID in URL vs hidden parameter

Can you think of a downside of a session ID in the URL?

If you give a link with your session ID to someone else, then that person might continue with your session!

Also, bookmarking a URL incl. the session ID does not (or should not) make sense, as the next time you use the bookmark you should start a different session

Solution 3: sessionId in a cookie

Standard solution built into HTTP and browser

- Cookie is piece of information that is **set by the server** and **stored by the browser**
 - namely when HTTP response includes Set-Cookie field in header
 - It belongs to some domain, eg `www.test.com`
 - It includes **expiry date**, **domain name**, optional **path**, optional **flags**
 - eg `secure` , `HTTPOnly` , and `SameSite` flags
- Cookie is **automatically** included in any HTTP request by the browser, for any request to that domain
 - in the `Cookie` field of HTTP request
- Cookie can include any type of information
 - sensitive information, such as **session ID**
 - less sensitive information, such as language preferences

Example cookie traffic

- **Setting** a cookie set with an HTTP response

```
HTTP/1.0 200 OK
```

```
Content-type  text/html
```

```
Set-Cookie: language=Dutch
```

```
Set-Cookie: sessionID=123; Expires=Tue, 26 Apr 2021 11:30:00 GMT
```

```
...
```

- **Sending** a cookie in an HTTP request

```
GET someurl.html HTTP/1.0 200 OK
```

```
Host: example.com
```

```
Cookie: language=Dutch, sessionID=123
```

Different types of cookies

- **non-persistent** aka **session cookies**
 - only stored while current browser session lasts
- **persistent cookies**
 - preserved between browser sessions
 - useful for maintaining user preferences across sessions
 - lousy for privacy...

Domains, subdomain, and top level domains

The domain in a cookie can be a **subdomain** of a website,

eg `cs.ru.nl` is a subdomain of `ru.nl`

Are cookies for `cs.ru.nl` sent with requests to `ru.nl` and `vv`?

Can `ru.nl` set a cookie for `cs.ru.nl` ?

Complex rules restrict cookie access across (sub)domains [RFC 6265]

Overall rationale: **subdomains need not trust their superdomain**

- Subdomains can *access* cookie for domain, but not vice versa
- Subdomains can *set* cookie for direct superdomain, but not `vv`
- With the `HostOnly` flag, cookies can further restrict access

For details, check [RFC6265] and hope browsers do not still implement parts of [RFC 2109] or [RFC 2965].

For **top level domains**, eg `.nl`, there are additional rules, to prevent `ru.nl` from setting a cookie for `.nl`

But does all this work as intended for countries that use 3 level domain names?

Eg for `somecompany.co.uk`, where `co.uk` is not a top level domain

Different ways to provide session ID

1. Encoding it in the URL

Downsides: 1) stored in logs (eg browser history), 2) can be cached & bookmarked, 3) visible in the browser location bar.

2. Hidden form field

Better: won't appear in URLs, so cannot be bookmarked, and less likely to be logged

3. Cookies

Best choice: automatically handled by browser; easier & more flexible.

But: such automation may have downsides...

Some session attacks

Aim of attacker: **get the session ID**

- This can be session cookie, or other form of session ID
- If the victim is logged in, this is just as good as stealing his username and password!
- **Stealing the session ID**
 - eg sniffing network traffic
(or XSS attacks discussed later)
- **Session Prediction**
 - try to guess a session ID
- **Brute Force**
 - try many guesses for the session ID, until you get lucky
- **Session Fixation**
 - make the victim use a known session ID

Session ID prediction attack

Suppose you can check your grades in blackboard on page
`brightspace.ru.nl/grades.php?s=s776823`

Is this a security problem?

If s776823 is your student number and also the session id
(in the URL in this case) then it is!

Attacker could try other student IDs or – better still – the
university employee number of a teacher.

Session fixation attack

If the sessionID is in the URL, an attacker can

1. start a session with say bank.com and obtain a session ID;
2. craft a link with that session ID and gets victims to click it, by
 - a) emailing victims with that link in the email; *or*
 - b) luring victims to a webpage with that link
3. The victim now goes to the website using a known session ID;
4. If victim logs in, and *session ID is not changed*, then attacker can join the session & abuse the user's rights!

Therefore: web server should **change session ID on login actions**

If the session-ID is a hidden form field, it does not end up in URL, attacker cannot email a link, but option 2b) is still possible, with a POST request.

Variant: attacker has already logged in, so victim joins the attacker's session and may enter confidential data (eg credit card number) for attacker's account.

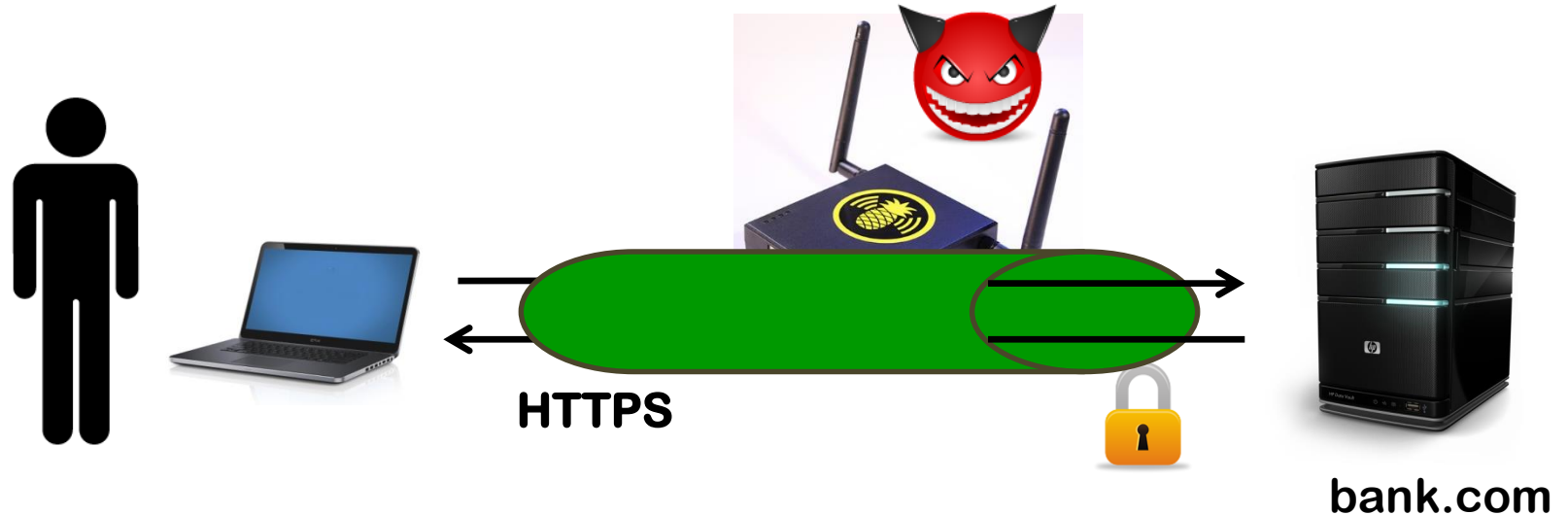
[aka **Login CSRF**, see *Surviving the Web* §3.3.4]

Making attacks on sessions harder

- Use long enough, random session IDs – ie with enough entropy
 - prevents session prediction and brute forcing
- Change session ID after any change in privilege level
 - eg after logging in
 - prevents session fixations
- Expire sessions
 - eg by setting expiration on cookies
 - reduces the attack surface in time
- Use HTTPS
 - for *all* requests & responses that include session ID, not just the login
 - prevents networking sniffing of session ID
- Let clients re-authenticate before important actions
 - reduces the value of any stolen session ID

Cookie stealing by MitM without breaking TLS tunnel

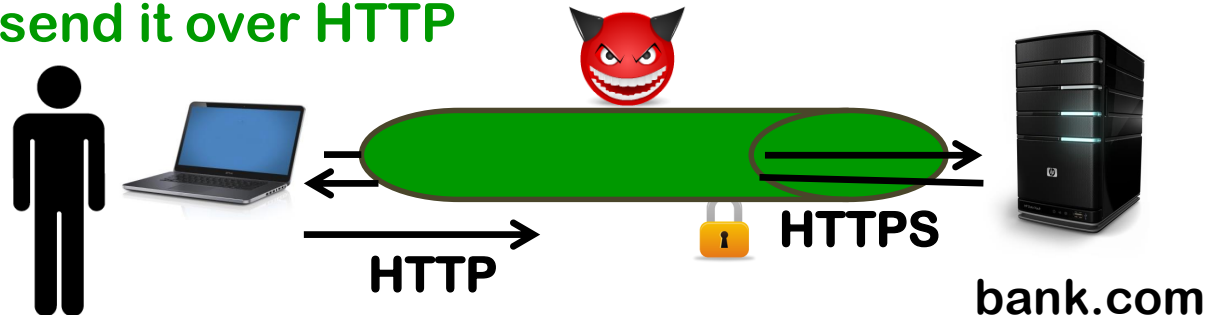
Without breaking the HTTPS tunnel, a Man-in-the-Middle (MitM) attacker may be able to steal cookies



Stealing cookies without breaking the TLS tunnel

1. User logs on to `https://bank.com`
2. Server sets session ID for `bank.com` in cookie
 - which is encrypted in HTTPS-traffic, so attacker cannot steal it
3. User does his banking
-
4. User asks for unencrypted HTTP link (eg `http://nu.nl`)
5. MitM attacker replies with a redirect to `http://bank.com`
6. Browser follows redirect and sends the bank's cookie over HTTP
7. Bingo! Attacker has the cookie

Solution: cookie has be declared as **secure**, which **disallows the browser to ever send it over HTTP**



Stealing cookies with scripts (XSS)

- If the attacker can sneak a script into the bank's website, the script can inspect the cookies and send them anywhere on the internet, including to <https://mafia.com>
- Such an attack, called **Cross Site Scripting (XSS)** attacks will be discussed in 2 weeks
- Solution: cookie has be declared as **HttpOnly**
 - This means the cookie is **only used for sending along with HTTP requests & never allowed to be accessed by scripts running in the webpage**

Abusing cookies without stealing them (CSRF)

Suppose webpage `bank.com` contains

```
<a href="transferMoney?amount=1000  
      &toAccount=52.12.57.762">
```

Suppose attacker sets up a malicious website `mafia.com` with

```
<a href="https://bank.com/transferMoney?amount=1000  
      &toAccount=52.12.57.762">
```

Will there be a difference in HTTP requests?

In other words, can bank.com tell the difference?

- No!
- The browser automatically attaches the bank's cookies to both requests! So if the victim is logged in to the bank, money will be transferred
- This is called a **Cross-Site Request Forgery (CSRF)**

Countermeasures against CSRF

1. Let client re-authenticate before important actions
 - eg. when resetting their password
2. Anti-**CSRF token** aka **Tokenization** [See **Surviving the Web §4.4.3**]
 - the bank always includes unpredictable **CSRF token** in any webpage it creates, which is added as hidden parameter in all requests
 - effectively a second session ID
 - Only links from a legitimate bank.com webpage will have the right value for this token;
Cross-site requests from mafia.com may have the right cookie but not the right token value
3. set **SameSite** flag of the cookie (New since Nov 2017)
 - **strict** cookie never attached to cross-site requests
 - **lax** cookie only attached to top-level GET requests
i.e. GET requests which change the address bar to bank.com
(so not for loading an iframe on mafia.com)
4. Newer still (2019) Let browser add **Set-Fetch-Site** header to distinguish cross site requests (no exam material)

Making attacks on session cookies harder

Three defense mechanisms

1. `secure` - only ever sent cookie over HTTPS, never over HTTP

Encrypting the cookie itself, when it is sent over HTTP, is pointless.

Why?

Attackers can simply replay a stolen encrypted cookie!

2. `HttpOnly` - inaccessible to scripts

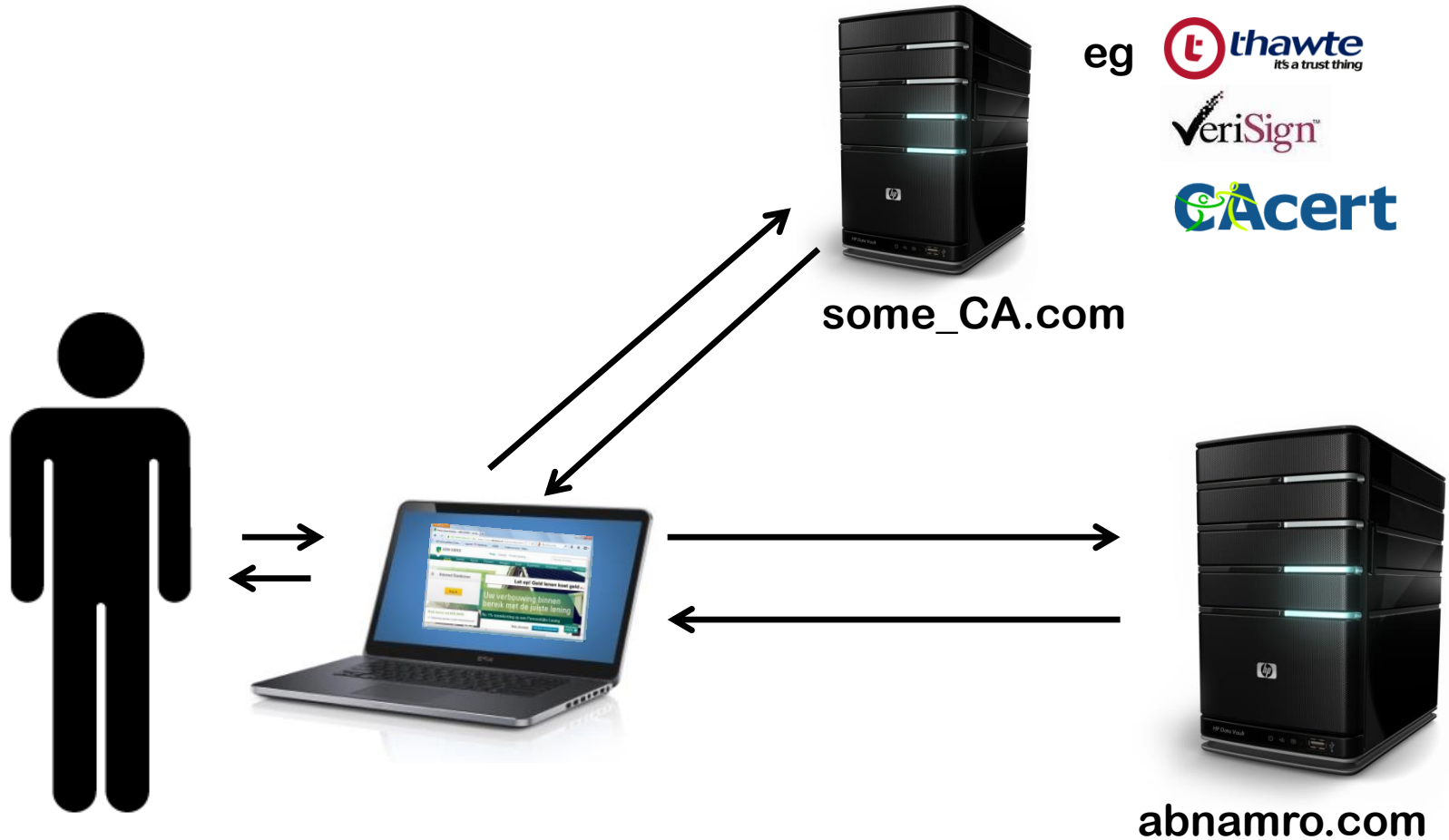
3. `SameSite` – not usable from links on third party sites

NB these mechanisms **protect against different types of attacks:**

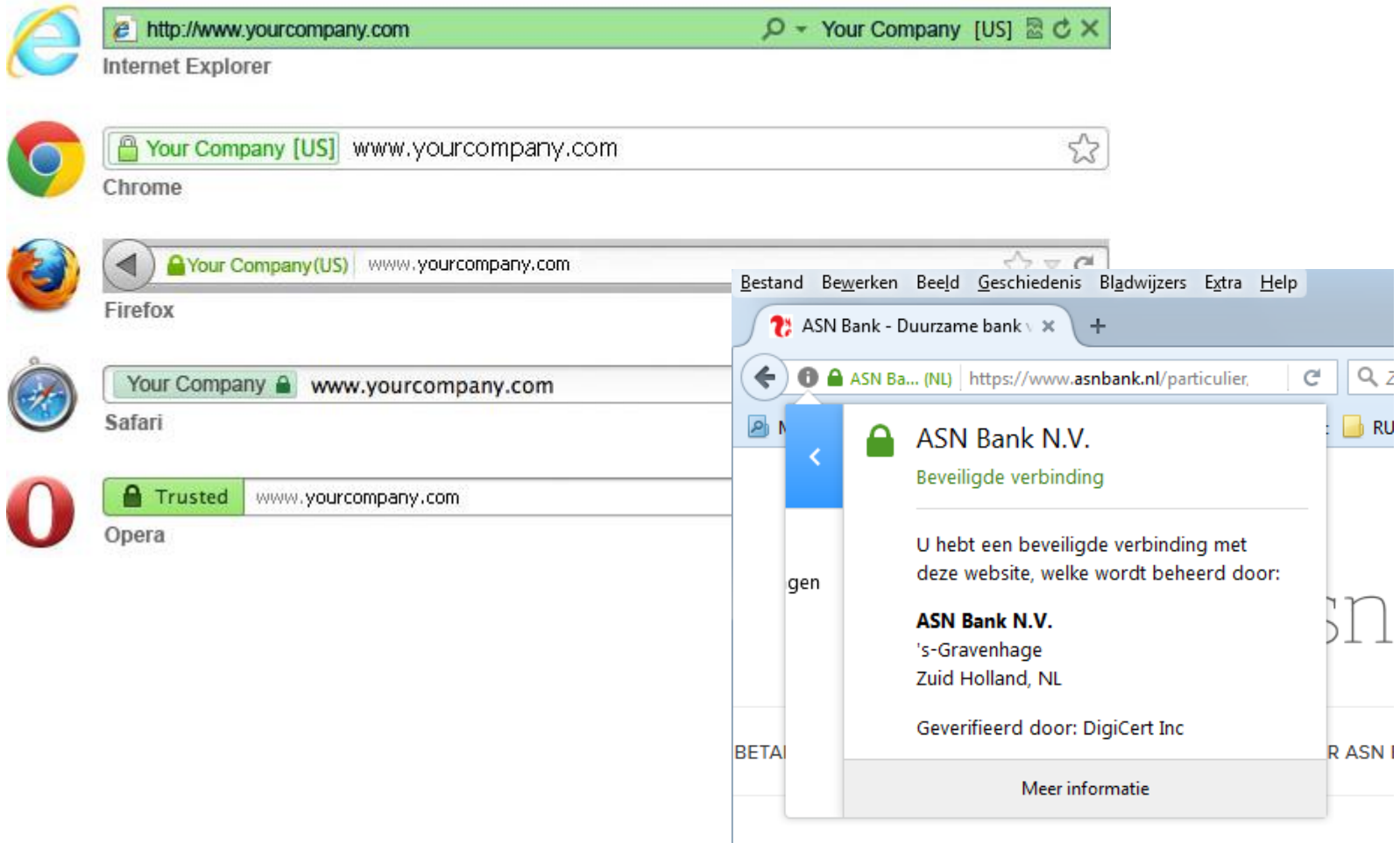
1. protects against **eavesdropping**
2. protects against **client-side scripts** injected into the ‘real’ website (discussed in later lecture)
3. protects against **malicious sites**

Attacking HTTPS

What are we trusting?



EV certificates recognisable in browser



DV, OV and EV SSL certificates

CAs validates *who* is requesting a certificate for a domain in different ways:

- **DV (Domain Validation)** certificates
 - Email check to validate that this is the owner of that domain, using whois information (eg via <https://www.sidn.nl/whois>)
 - Free via **Let'sEncrypt** since April 2016
- **OV (Organisation Validation)** certificates
 - Additional check on identity & existence of the organisation eg against Chamber of Commerce records
- **EV (Extended Validation)** certificates
 - More rigorous check on identity of the organisation
 - How much extra security EV gives over OV brings is debatable...

Certificates can be **wild-card certificates**,
eg for `*.ru.nl` instead of `www.ru.nl`

Trusted Computing Base (TCB)

The **Trusted Computing Base (TCB)** is the smallest amount of software (and hardware, people, organisations, ...) that you **MUST** trust for some security property.

The TCB for HTTPS is huge: we trust that

- the TLS software is correct (but: HeartBleed)
- the web-server protects its private key
- the browser checks the certificate correctly, including the CRL
- the Certificate Authorities (CAs)

For Firefox, <https://ccadb-public.secure.force.com/mozilla/IncludedCACertificateReport>

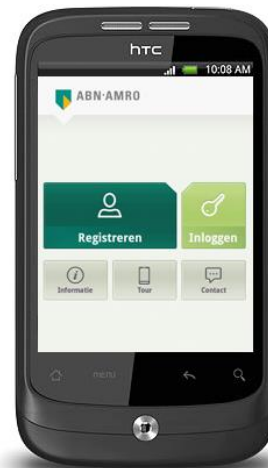
- the user checks for lock icon (and maybe the content of the certificate?)
- there is no malware in our browser or on our machine that is faking the display
-



Things that go wrong ...

March 2012: ING banking app does not check SSL certificate

December 2012: ABN-AMRO app does not check SSL certificate



Built-in HTTPS support in your browser is more likely to be correct than TLS support built into some app?

Things that go wrong ...

Certificate Authority DigiNotar was hacked in 2011.

Fake certificates for google.com were issued, presumably for use in Iran



DigiNotar provided *all* the certificates for the NL government...



Problem detected because the Chrome browser checks for suspicious certificates for google.com

Darknet Diaries has a nice podcast episode on this:
<https://darknetdiaries.com/episode/3/>

Countermeasures



- **HPKP (HTTP Public Key Certificate Pinning)**
 - browser remembers the public key of domains, and refuses new certificates for that domain with a new key
 - Not a big success: abandoned in 2017, because of the risks it introduces
 - eg if a wrong public key ends up in the browser, there is NO way to recover from this
- **Certificate Transparency (CT)**
 - Initiative to publicly share info on all issued certificates
 - See eg <https://crt.sh>
 - Organisations can spot if a rogue certificate is issued in their name: i.e. someone at Radboud should periodically (daily?) check there are no rogue certificates issued for ru.nl

Attacking HTTPS with Man-in-the-Middle attack

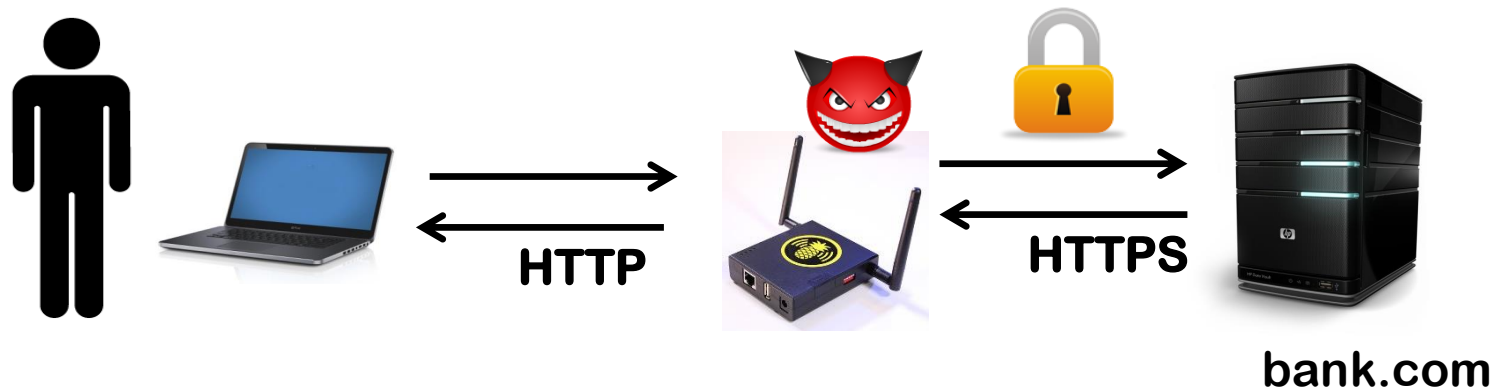
- *Capabilities* of the attacker
 - eavesdrop on traffic & modify traffic
(i.e. imagine of a proxy)
- *Goals* of the attacker
 - seeing plaintext data
 - modifying traffic without user noticing
 - or just: seeing the cookie

Different **attack vectors/attack scenarios** for a MitM attack:

- malicious wireless access point
- fake website, which redirects the traffic to the real website
- by the ISP, e.g. nation state intercepting all internet traffic

Simple SSL stripping : HTTP + HTTPS

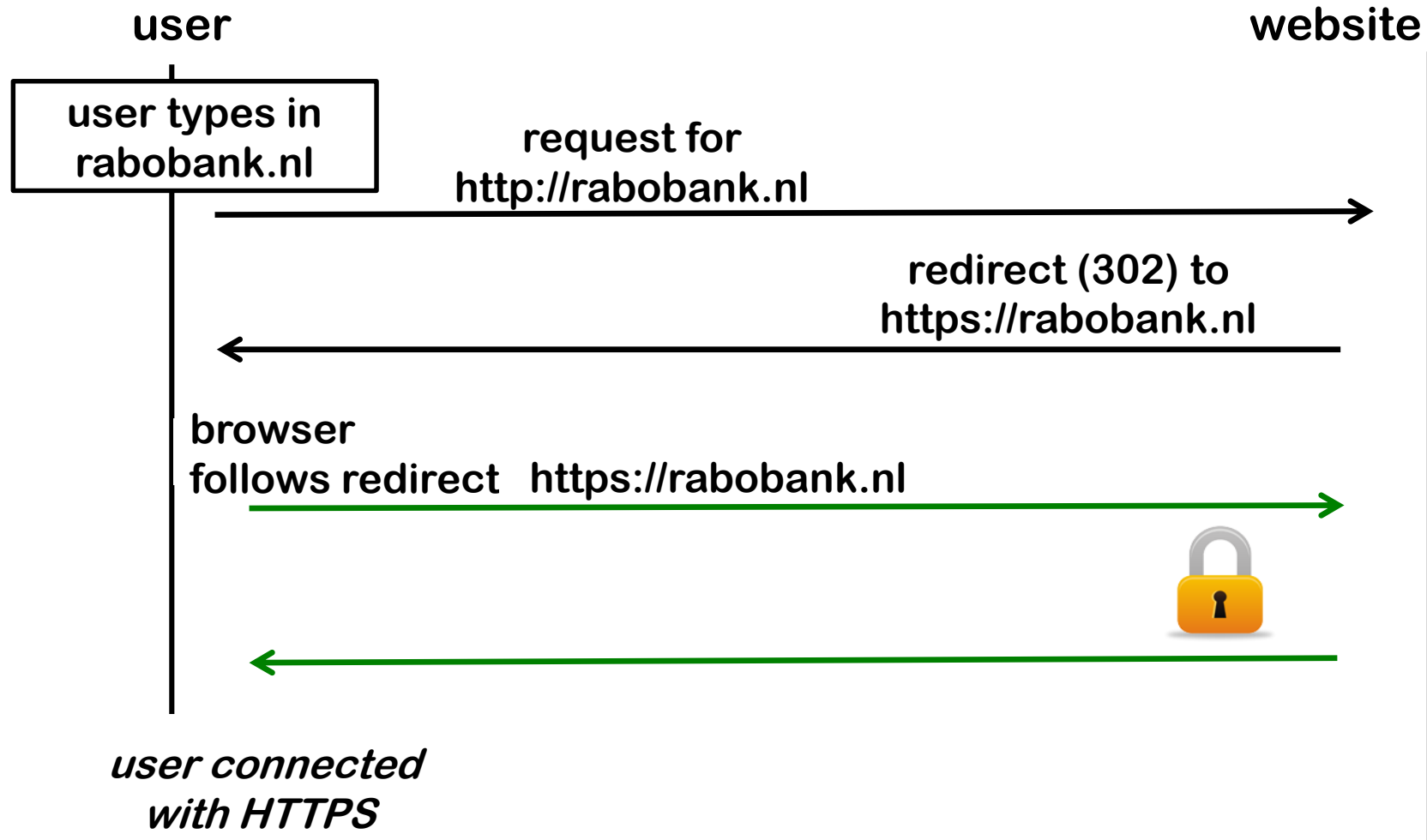
The idea: the attacker forces the browser to fall back to HTTP and hopes the user won't notice the missing **s**



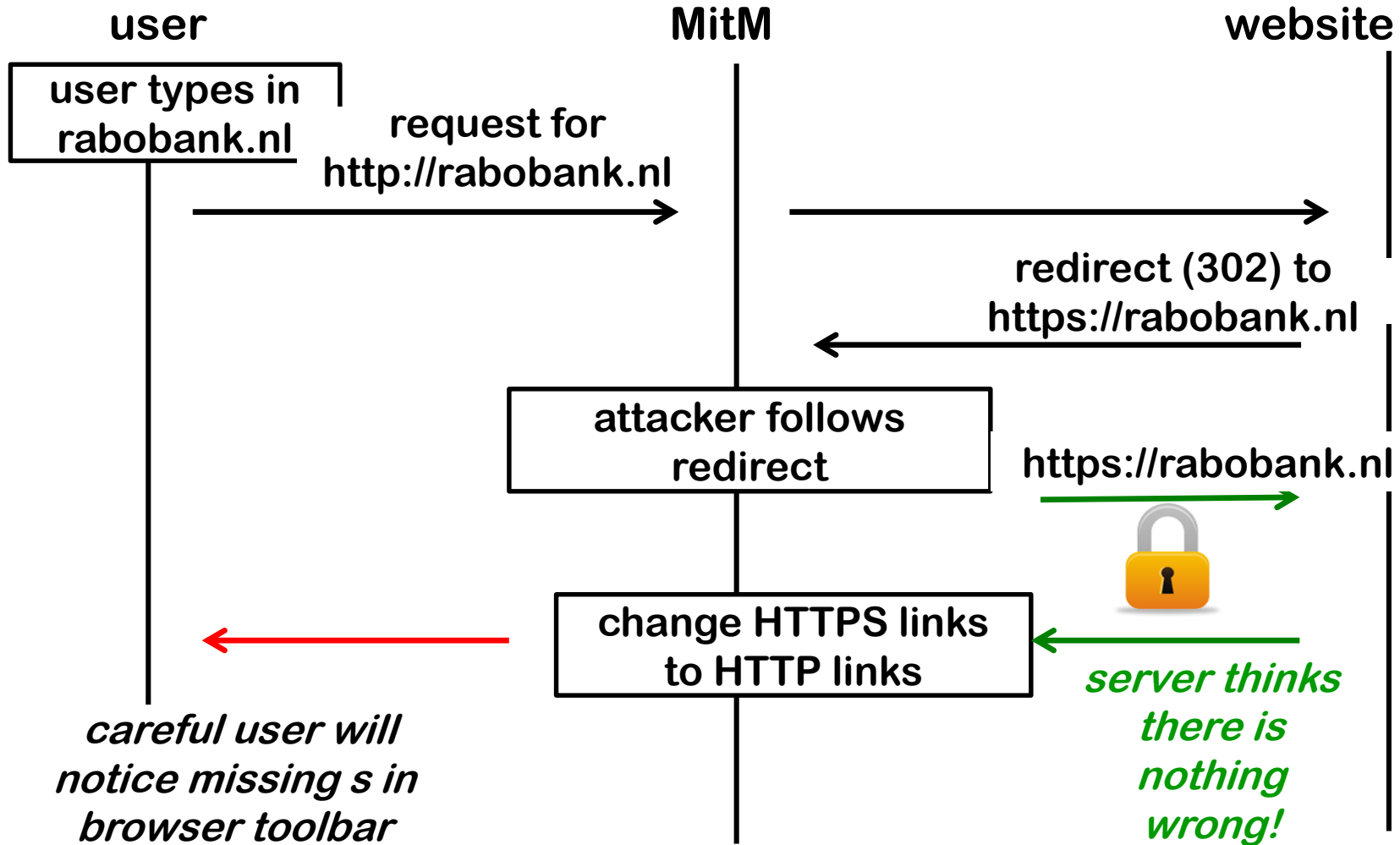
When can the attacker do this? If the user

- a) types in rabobank.nl, without https in front of it
- b) begins a HTTPS session by clicking on a link in a webpage that was retrieved with HTTP

Normal start of HTTPS session via HTTP request



MitM attack on this start of HTTPS session



Simple SSL stripping

- The MitM attacker
 - strips **S** from HTTPS in links in traffic from server to user
 - puts this **S** back in traffic from the user to the server
- The result



- The attacker can now intercept a username and password that the user sends
- After intercepting this information, the attacker could stop the MitM attack
 - and the user can then no longer see anything wrong!
- Attacker could also make arbitrary alterations to the web page

Won't secure cookies help?


- Secure cookies won't be sent by the client's browser over HTTP

But attacker can defeat this

by removing the secure bit from Set Cookie instruction
when forwarding traffic from the server to user

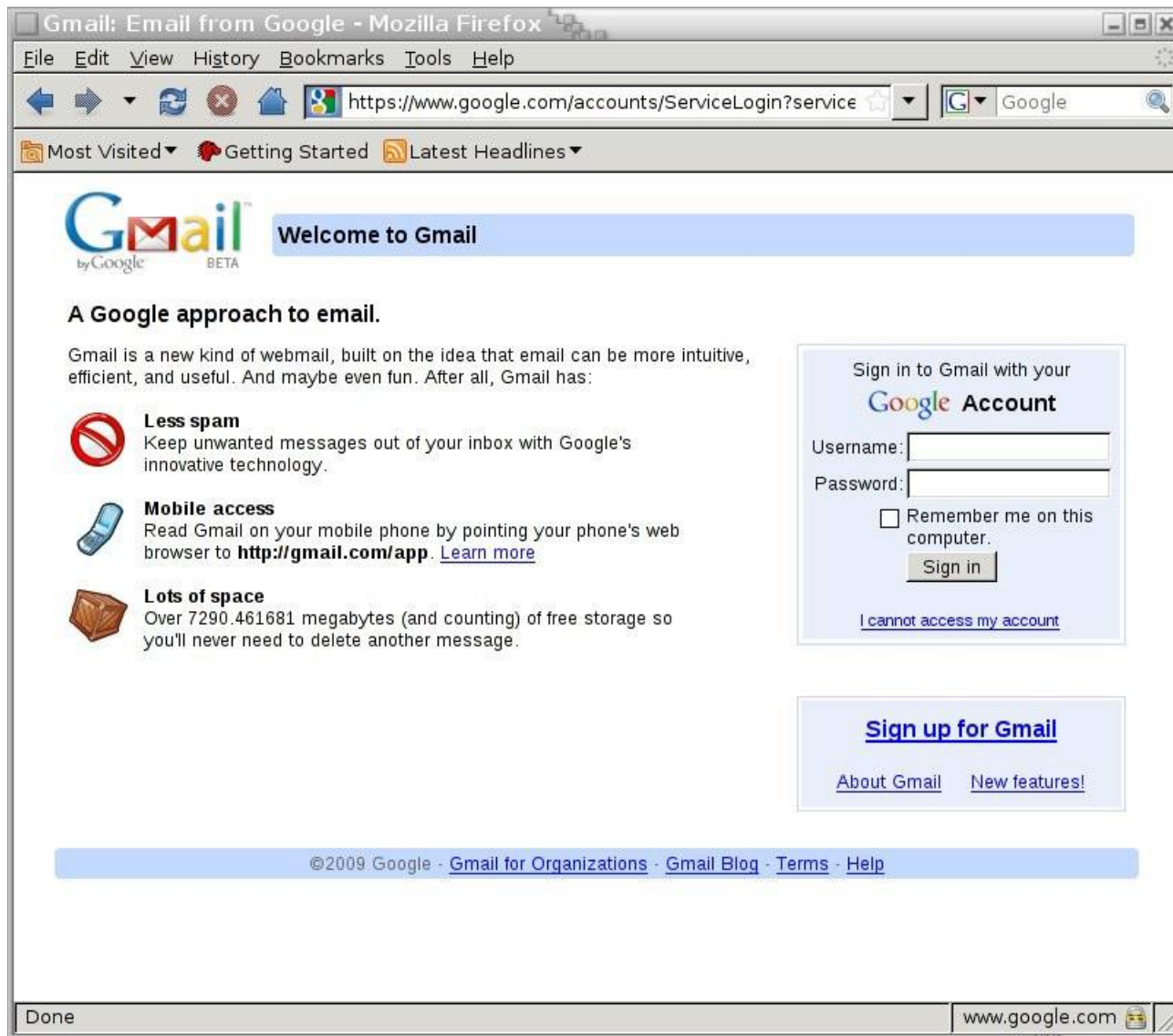
Spotting this attack?

A careful user can spot this attack

- the URL misses the s in https
- the little lock is missing in the browser corner 

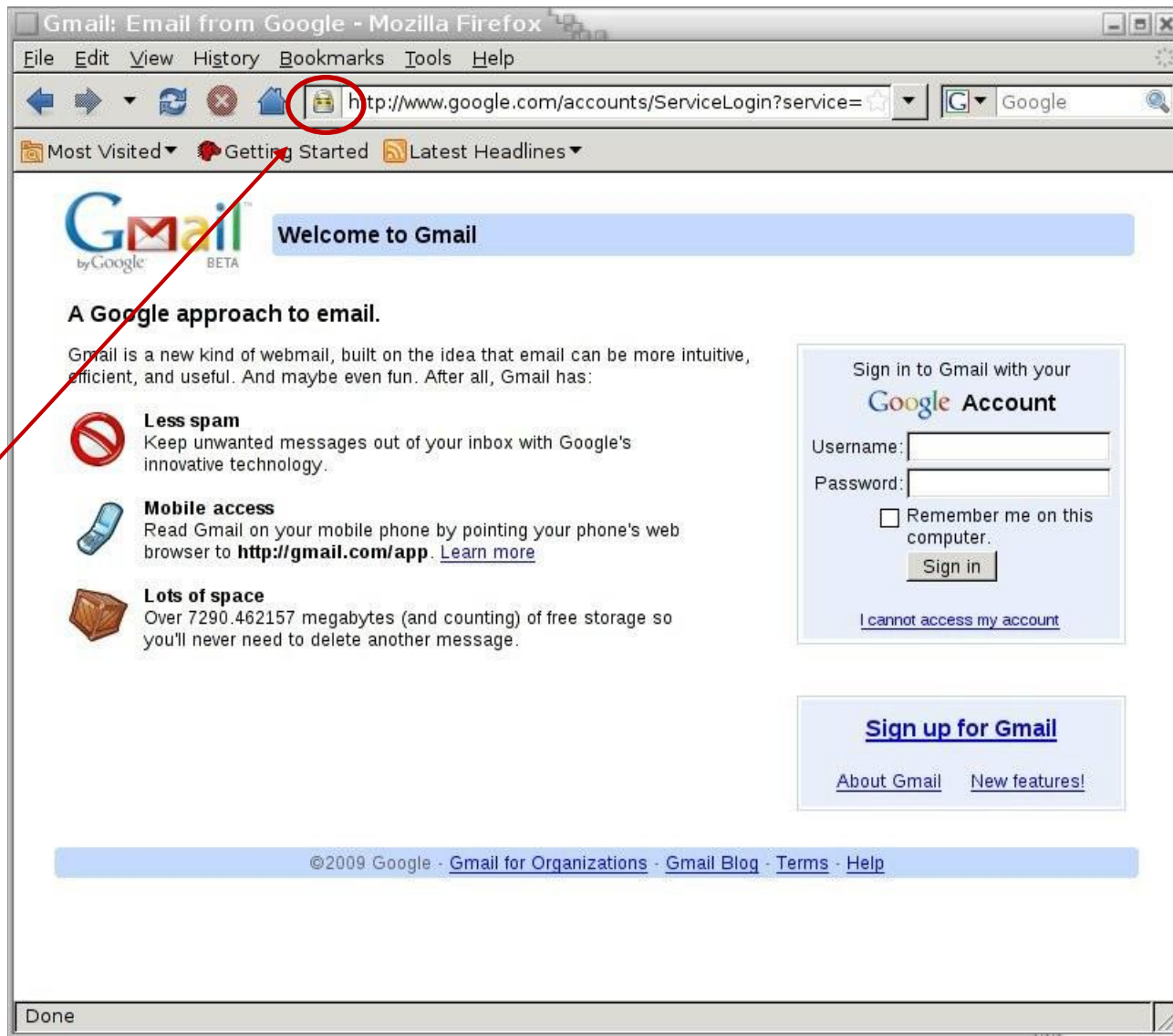
Nice improvement: the attacker can add  as favicon

The original secure site



SSL stripped version

lock
as
flavicon

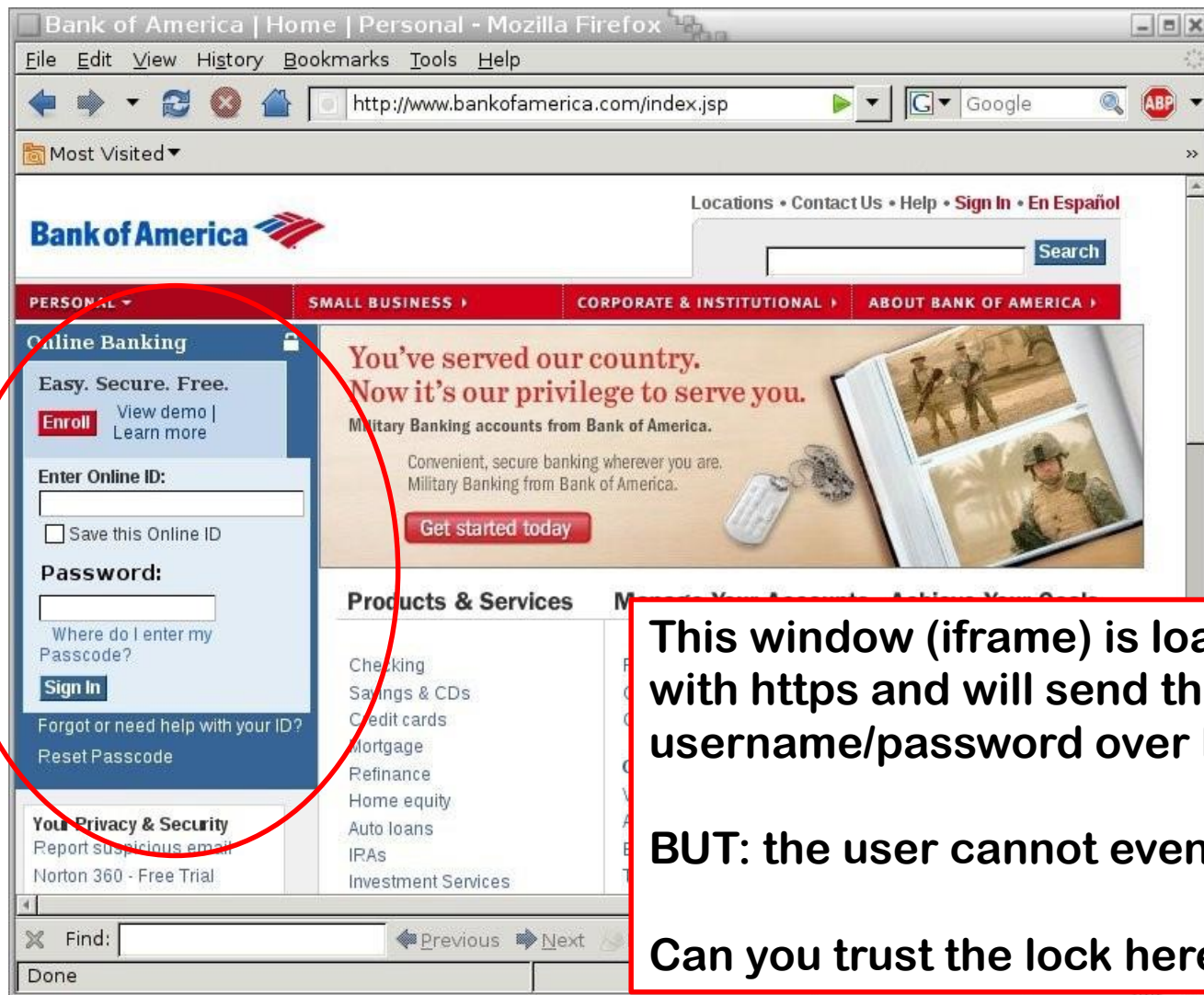


The original secure site



The SSL stripped version





Mixing http & https

Moral of the last example:

- Never use https for a frame inside a http page
- Never issue https requests from an http page

Web browsers nowadays warn about (or even block) mixed http/https content.

Demo: check out how this works in your browser

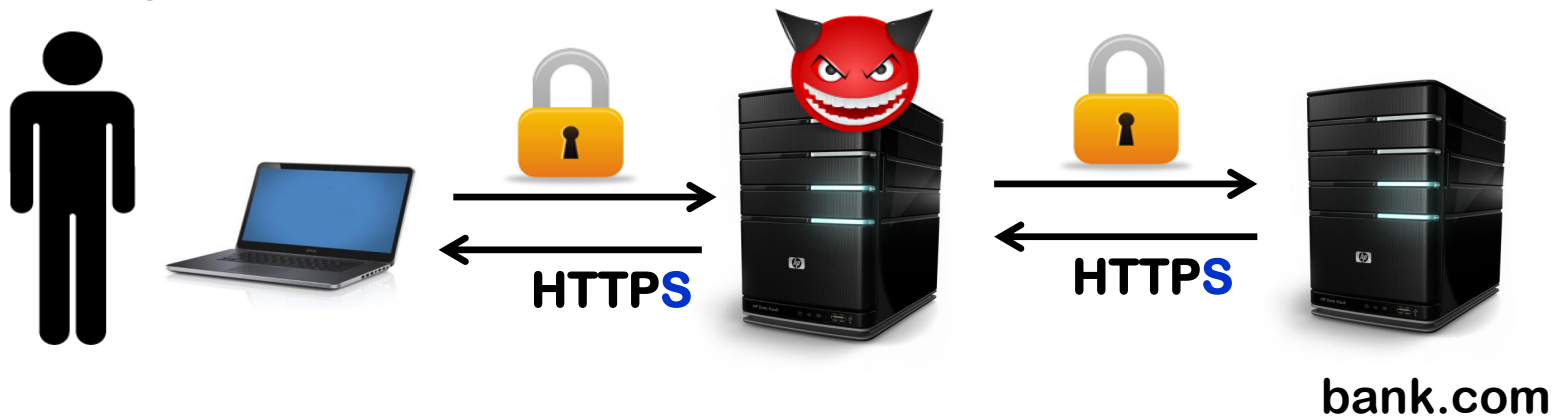
http://www.cs.ru.nl/~erikpoll/websec/demo/mixed_content.html

https://www.cs.ru.nl/~erikpoll/websec/demo/mixed_content.html

(Advanced SSL stripping – not exam material !)

Can SSL stripping can be improved so that the user sees a HTTPS session?
Sometimes it can, by exploiting browser and certificate bugs!

Resulting situation



To see how, watch Moxie Marlinspike's talk at DEFCON 17
<https://www.youtube.com/watch?v=ibF36Yyeehw>

As countermeasure this and other SSL strip variants, HSTS was introduced

Other countermeasures to SSL stripping

- **HTTPS Everywhere** browser plugin
 - ie. simply never use HTTP
- **HSTS (HTTP Strict Transport Security)**

HTTP Strict Transport Security (HSTS) [RFC6797]

Protection against SSL stripping

1. website (e.g. bank.nl) tells the browser that it only ever wants to use HTTPS, in HTTP response header

```
Strict-Transport-Security: max-age=15768000;  
includeSubDomain
```

2. the browser remembers this, and will in future turn http requests for that domain into https requests

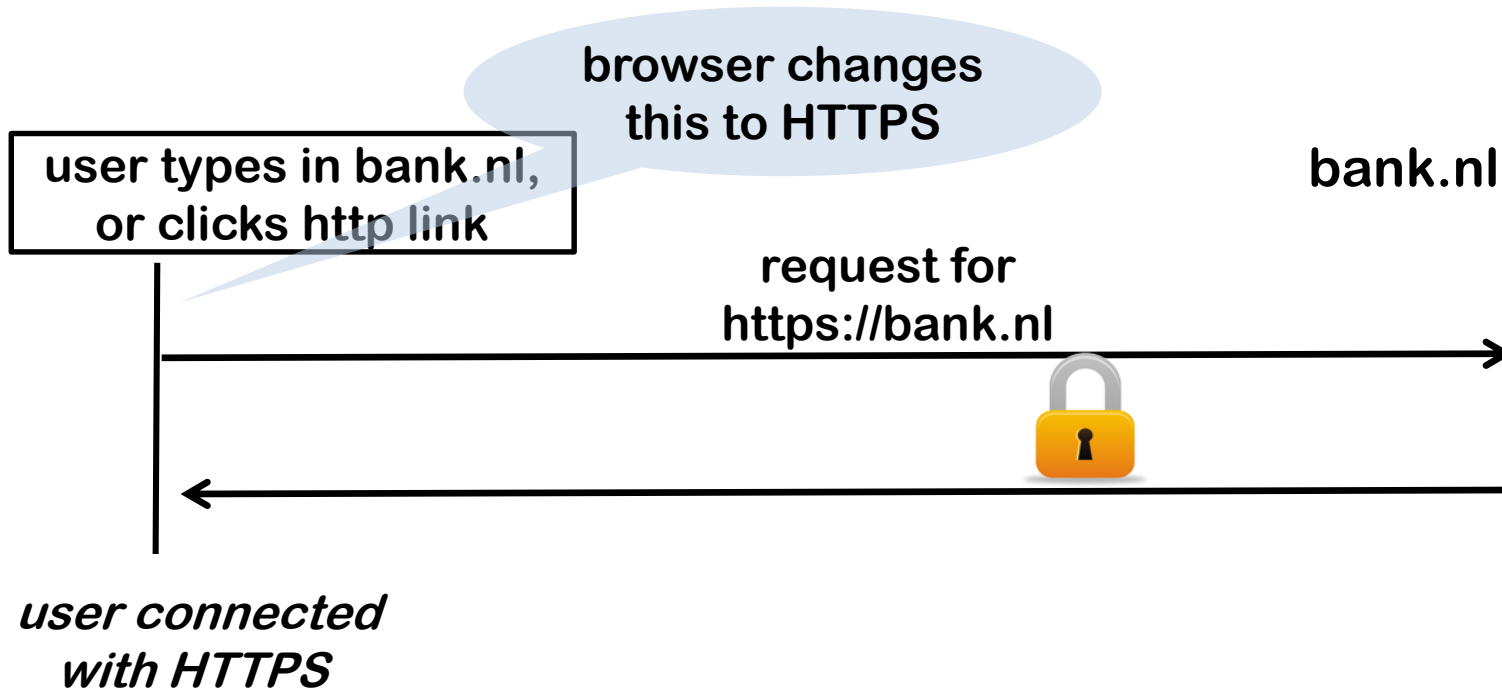
Eg browser will turn `http://bank.nl/rekening/...`
into `https://bank.nl/rekening/...`

HSTS is now supported by all mainstream browsers.

HSTS

On very first visit to bank.com, the browser stores some information, recording that bank.com wants to talk HTTPS only.

For subsequent visits



Checking for HSTS usage

- In browser

- In Firefox:

- type `about:support` in the address bar. In the Application Basics section, you will see Profile Folder. Click Open Folder, and look for file `SiteSecurityServiceState.txt`

- In Chrome:

- type `chrome://net-internals/#hsts` in address bar

- In HTTP traffic:

- look for HSTS field in HTTP header, of the form

- `Strict-Transport-Security: max-age=15552000; preload`

- On Linux, with `curl -si "https://www.ru.nl" | grep Strict`

Remaining problem with HSTS

- There remains risk with the *very first* request to a site
 - a MitM attacker could SSL strip the first request, and not remove the HSTS header from the HTTP response
- Solution **HSTS preload list**: browsers come with preloaded HSTS info for some sites, so even the very first requests to these sites will never be with HTTP
 - See <https://hstspreload.org/>
- There remains a privacy risk with HSTS: the HSTS info stored in your browser can reveal which sites have been visited... even if you do this in private browsing mode?

Exercises for this week

A. Checking input sanitisation in Brightspace

How is input encoded & sanitised in Discussion Forums, at the client side and/or at the server side?

B. Checking security settings (cookie setting & HSTS, Certificate Transparency, ..) for some sites where you have a login

C. One more WebGoat lesson

Authentication Flaws - Authentication Bypasses

**NB A & B to be handed in (in pairs) via Brightspace
Deadline Friday 11 Sept, 23:59**