

WEB APPLICATION SECURITY



LESSON 1: BASICS

LEARNING OUTCOME

On completion of this module, students will be able to:

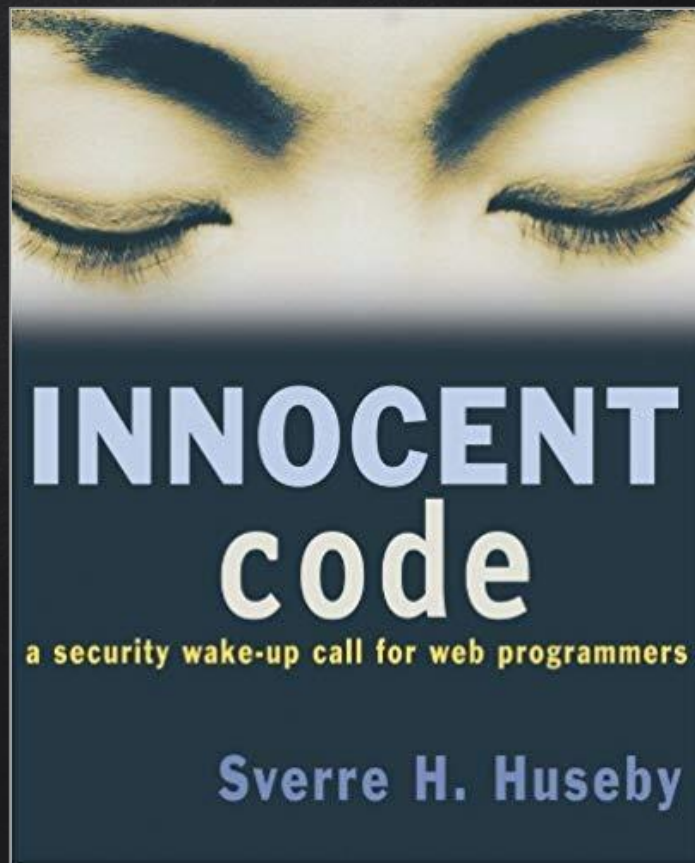
- 1) Understand **web application security** and its importance.
- 2) Understand **common mistakes** of coders and **vulnerabilities** of web applications.
- 3) Explain how code developers' mistakes may **be exploited** to the benefit of the attackers and how to **prevent** these attacks.
- 4) Build **secure web applications** using secure coding practices.

TEXT BOOK

- *Innocent Code: A Security Wake-Up Call for Web Programmers*

Wiley; 1 edition (January 30, 2004)

By Sverre H. Huseby



EXTRA READING

- Web Security Testing Cookbook:
Systematic Techniques to Find
Problems Fast

O'Reilly Media; 1 edition

By Paco Hope, Ben Walther

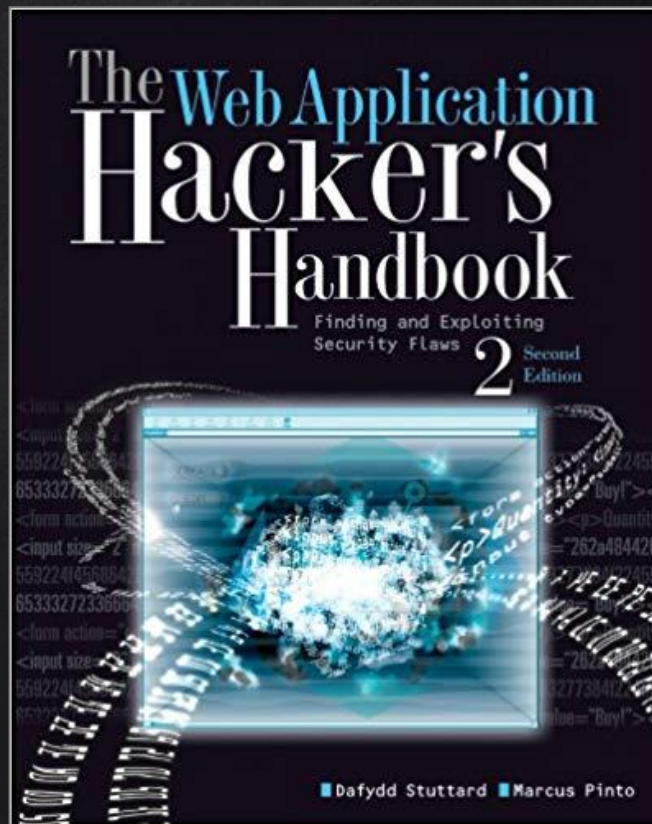


EXTRA READING

- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws

Wiley; 2 edition (September 27, 2011)

By Dafydd Stuttard, Marcus Pinto

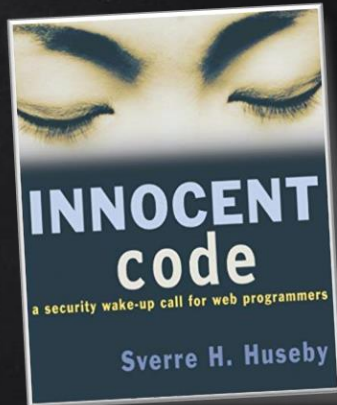


ASSESSMENT

Final Exam: 100%

The exam material covers the contents of the book "Innocent Code", supplemented with variations on it that are addressed in the weekly classes and labs.

The minimum pass mark is 60%.



- You understand that in this class we may cover **methods to exploit vulnerabilities** in contemporary computer systems and computer networks.
- You further understand that we may learn **techniques employed by unethical individuals** to **circumvent** security mechanisms, **violate** copyright, cause **damage**, cause **financial loss**, or **break** the law in other ways.
- You hereby pledge to use all information obtained in this class in an **ethical** and **responsible** manner, properly observing University Policy and the law.

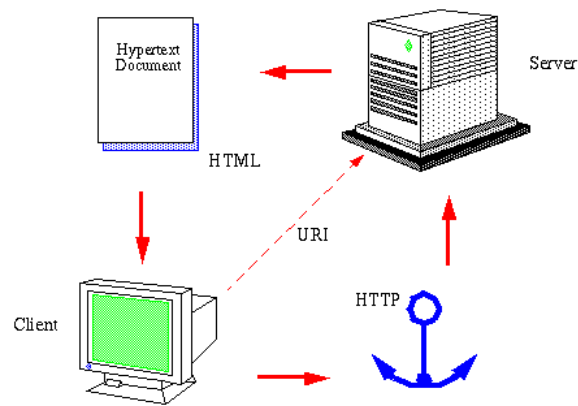
DISCLAIMER

- You further pledge to **abide** by course rules, in particular (but not limited to) hacking systems declared off-limits by the instructional staff.
- By using the course materials, you accept that you will **only lawfully use it** in a **test lab** ~ with devices that you own or are allowed to conduct penetration tests on ~ to enhance your own knowledge.
We **do not endorse** use of any information expressed on the course **outside** of a lab environment.
- Any actions and or activities related to the materials contained in this course is solely **your responsibility**. The **misuse** of the knowledge and information in this course can result in **criminal charges brought against** the persons in question.

INTRODUCTION

Three specifications are central to the Web:

1. **URL** (Uniform Resource Locators)
[URI: Uniform Resource Identifiers]
2. **HTML** (HyperText Markup Language)
3. **HTTP** (Hypertext Transfer Protocol)



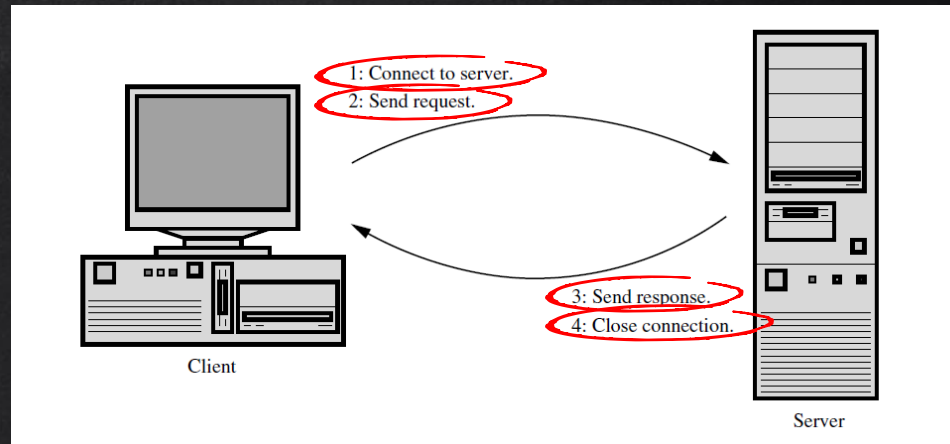
HTTP

(HYPERTEXT TRANSFER PROTOCOL)

CLIENT-SERVER MODEL OF WEB

TCP **connection** is established, the browser sends a **HTTP request** asking the web server to provide the wanted document.

The web server sends a **reply** containing the page contents, and **closes** the connection.



The **browser** is always the **initiating** party—the server never “calls back”.

This means that HTTP is a **client/server protocol**.

The client will typically be a web browser, but it need not be. It may be any program capable of sending HTTP requests to a web server.

REQUESTS AND RESPONSES


HTTP is **line oriented**, just like many other Internet protocols.

Communication takes place using **strings of characters**, separated by **carriage return** (ASCII 13) and **line feed** (ASCII 10).

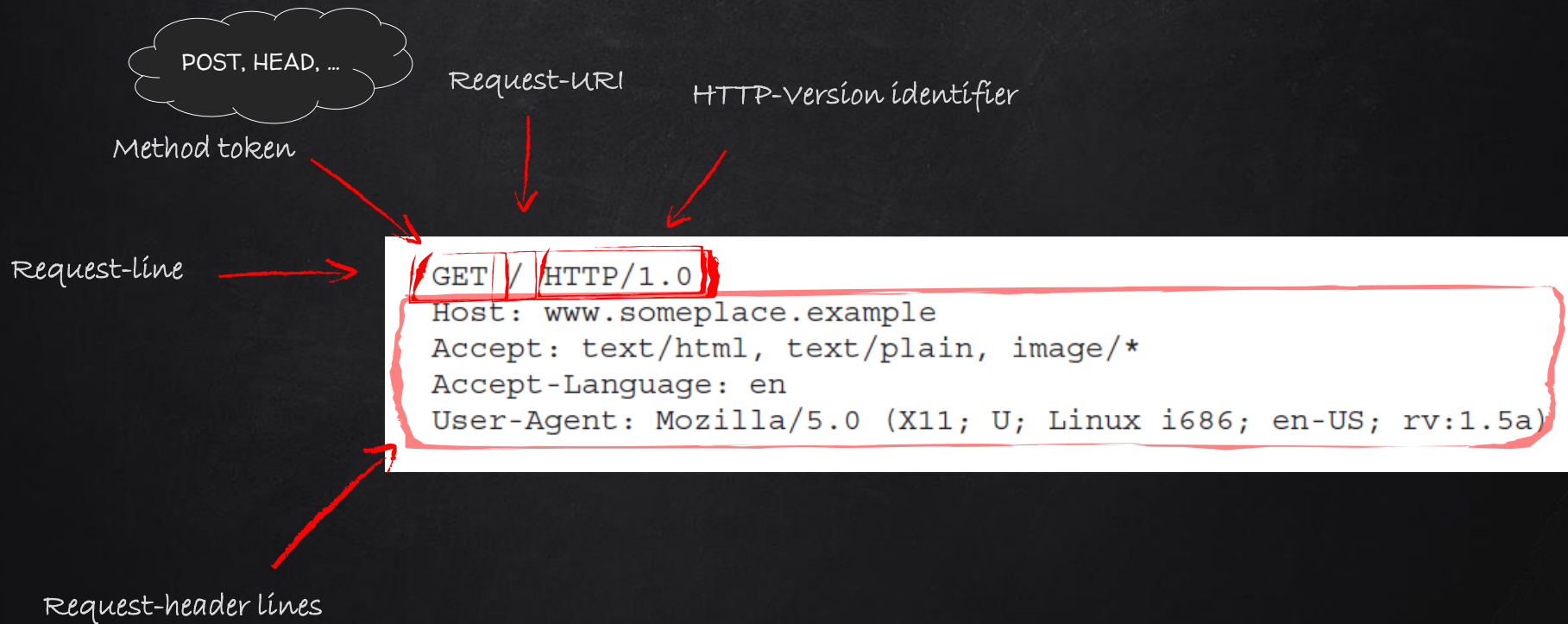
Example: `http://www.someplace.example`

```
GET / HTTP/1.0
Host: www.someplace.example
Accept: text/html, text/plain, image/*
Accept-Language: en
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a)
```

Request



GET REQUEST



"404 Not Found"

HTTP-Version

Status-Code

the Status-Line

HTTP/1.1 200 OK

Date: Sun, 07 Dec 2003 21:16:12 GMT

Server: Apache/1.3.27 (Unix) PHP/4.3.2

Last-Modified: Wed, 20 Aug 2003 20:31:11 GMT

Content-Length: 84 Bytes

Connection: close

Content-Type: text/html

Request-header lines

<html>

<head><title>Test</title></head>

<body>

<p>Hello, world!</p>

</body>

</html>

GET REQUEST VS. POST REQUEST

POST requests should be used when the action about to be taken has **side effects on the server**, i.e. when something is permanently changed.

- With GET, a client **asks** for information.
- With POST, the client **contributes** information.
- With GET requests, the browser is **free to resend** the request, for example, when the user presses the "**back button**" in his browser (not suitable for money transfers in a bank).
- POST requests, on the other hand, **cannot be reissued** by the browser without first asking the user for **permission** to do so.
- In a GET request, any parameters are encoded **as part of the URL**.
- In a POST request, the parameters are **"hidden"**.

POST REQUEST EXAMPLE

```
POST /login.php HTTP/1.0
Host: www.someplace.example
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a)
Referer: http://www.someplace.example/login.php
Content-type: application/x-www-form-urlencoded
Content-length: 49

username=jdoe&password=BritneySpears&login=Log+in
```

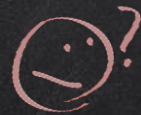
The parameters are **encoded** as you are used to, but they are **hidden in the request** rather than being part of the URL.

- URL Encoding refers to the **escaping** of certain characters by encoding them using a percent sign followed by two hexadecimal digits.

AT&T → AT%26T

SECURITY CONCERN

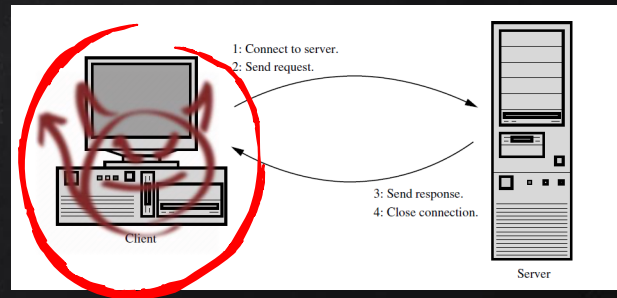
What can be the security concern?



As all requests originate on the **client-side**, that is, on computers of which the user has full control, nothing stops the attacker from **replacing** the browser with something completely different.

As HTTP borrows its line oriented nature from the **telnet protocol**, you may actually use the telnet program to connect to a web server.

```
telnet www.someplace.example 80
```



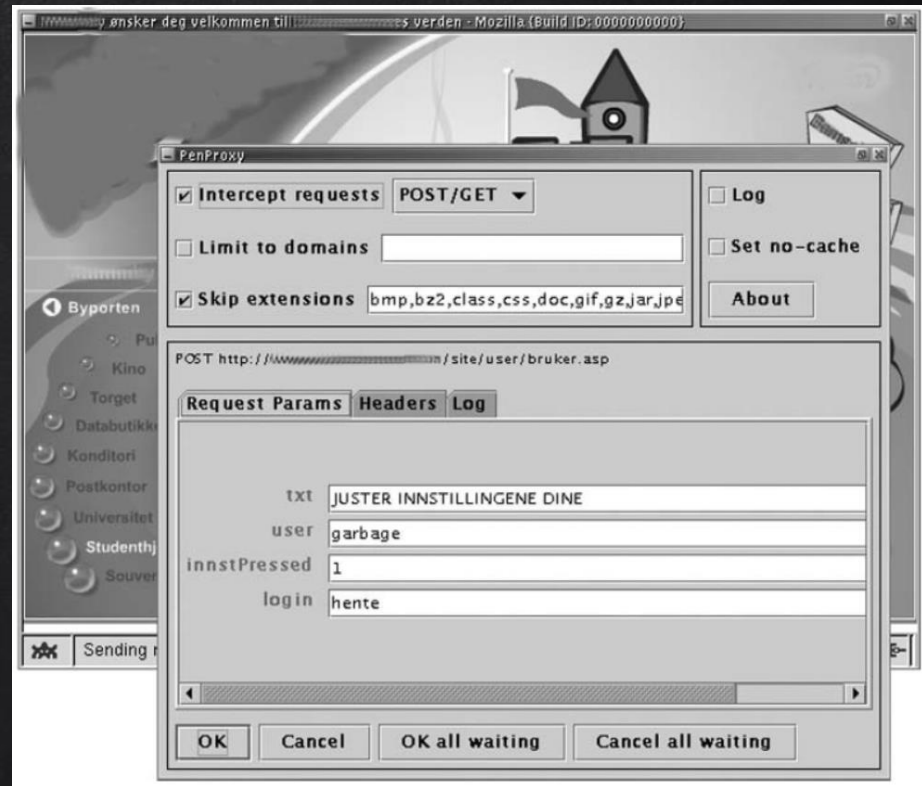
⚠️ Everybody can **write a program** to connect a socket and do the actual protocol conversation for him/her.



There are freely available programs that will aid them in **manipulating** all data that get sent to the server.

Some of these programs are **proxies** that sit between your browser and any web server, and that pop up nice dialogs whenever your browser sends anything.

The proxies let you **change headers and data** before they are passed to the server.



THE REFERER HEADER

A Referer header is sent by most browsers on most requests.

- The header contains the URL of the document from which the request originated.

Example: `http://www.site.example/index.html`



```
  
<a href="http://www.news.example/index.html">News</a>
```

The HTML snippet includes an image from `www.images.example` and links to a page on `www.news.example`.



WHAT IS THE SECURITY CONCERN WITH REFERER HEADER?

20

When the browser views the HTML, it will immediately connect to www.images.example to obtain the image.

- When requesting the image, the browser sends a Referer header like this:

```
Referer: http://www.site.example/index.html
```

The URL **points to the page** from which the image was referred.

Any Java Applets, ActiveX, scripts and plug-ins included in the page would give the same Referer header.

And not only included objects: if the user clicks the link given above, www.news.example will receive the same Referer header.



Let's see it online: <https://www.hogeschoolrotterdam.nl/>

1. It **leaks** information to remote sites.

Any part of the URL, including parameters, will be visible to the third-party web server and any proxies that handle the request.

2. The Referer header originates on the **client**.

Some web sites check this header to make sure the request **originated** from a page generated by them (e.g. to prevent attackers from saving web pages, modifying forms, and posting them off their own computer).

- This security mechanism will **fail**, as the **attacker will be able to modify** the Referer header to look like it came from the original site.

Caching refers to **temporarily storing** documents close to the final destination, in order to reduce download times.

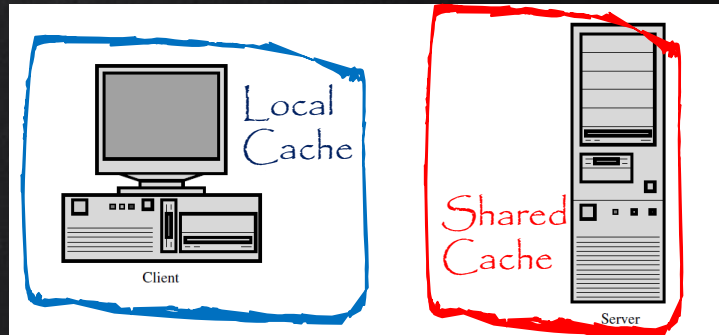
Two types of web caches:

- **Local** cache: is managed by the browser itself.

When the browser requests a document from a remote server, it often **stores a copy on the disk or in memory**.

- **Shared** (proxy) cache: is typically a server in the local area network.

If one user reads an on-line newspaper, and another user reads the same paper shortly after, the **proxy cache may serve a local copy** of the document to the second user.





Some documents **should not be cached**. For example:

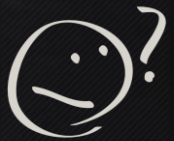
- Visitors a **stock information web site** most likely want **up to date** stock information, not yesterday's news.
- People might be able to use the **back button** to see other people's web pages in a shared browser, such as in Banks, Internet cafés.

Such sites **need a way to tell browsers and proxies** that documents should not be cached, or that they may only be cached for a limited time.

As with most other control information on the web, cache control is handled by **HTTP headers**.

Different versions of HTTP specify different mechanisms for cache control.

HTTP is a **stateless** protocol (no ties connecting different requests from the same client).

- We would like to have **state** between requests. What state for example? 
- Google search engine once used cookies to allow users (even non-registered ones) to decide **how many search results per page they wanted to see**.
- **Cookies** are introduced as an extension to HTTP to give us just the state.

With cookies, the web server asks the client **to remember** a small piece of information.

This information is passed back by the client on each subsequent request to the **same** server.

COOKIES

HTTP headers are used for both setting and returning cookies.

When server wants the client to remember a cookie, it passes a Set-Cookie header in the reply:

```
Set-Cookie: Customer="79"; Version="1"; Path="/"; Max-Age=1800
```

Name of cookie
and its value

To which parts of the document
hierarchy on this server
that cookie should be returned

How many seconds to
remember it

Cookies are returned using the Cookie header:

```
Cookie: $Version="1"; Customer="79"; $Path="/"
```

Drawbacks:

- Firstly, cookies may be **limited in size**, so space consuming states cannot be safely represented using cookies.
- Secondly, cookies are **handled on the client-side**, so we have to keep making sure that a misbehaving user doesn't change the state to his own liking.

Both limitations would be solved if we could keep the state information **on the server side**.

SESSIONS

SESSIONS (SESSION OBJECTS)

Sessions are **server-side** collections of variables that make up the state.

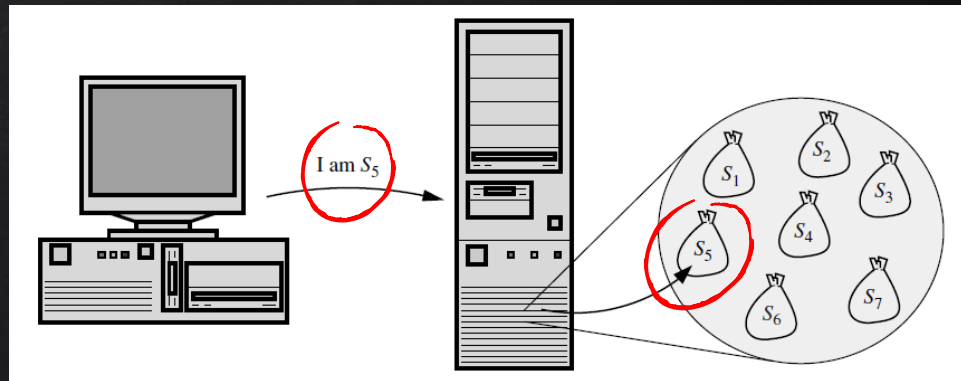
How to associate a set of data on the server to the correct client?

Answer: Session ID

- The common approach is to have the client pass a session ID on each request.

The session ID **uniquely** identifies one session object on the server, the session object “owned” by the **client making the request**.

☹ Security problem with sessions?

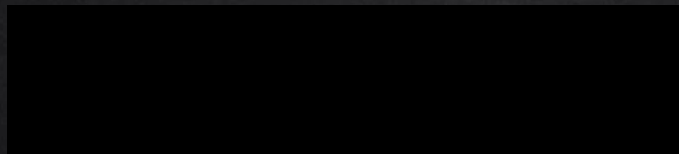


SESSION HIJACKING

Many web sites use a **session-based log-in**, in which a session is initiated once the user has given a valid user name and password.



If somebody gets **access** to the **session ID** of a logged in user?



The attacker would not need to know the password of the victim, as the session ID works as a “short-time password” or a **proof of successful authentication** after a user has logged in.

SESSION HIJACKING

How would the attacker gain **access** to the **session ID**?



- Guess it,
- Calculate it,
- Brute-force it,
- Find it by trial and error
- Cross-site Scripting
- Use referer header
- Packet sniffing
- ...



The security of sessions lay in the **secrecy** of the session ID.

The number one goal to prevent session hijacking is to keep the session ID **unavailable to third parties**.

As an **extra precaution**, many web sites implement **secondary measures** to limit the risk of session hijacking, even if a session ID becomes available to attackers.

However, **none** of these secondary mechanisms offer full protection against hijacking.

The **secrecy of the session ID** is the only mechanism that gives real protection.



1. Tie the session ID to the IP address of the client.

It will not protect against attackers who hide behind the same web proxy as the victim, as all requests from the same proxy will come from the same IP address.

2. Tie the session ID to certain HTTP headers passed by the client, such as the User-Agent header.

This approach isn't bulletproof either: an attacker could mimic the headers sent by several popular browsers. One of the combinations would probably let him through.



3. Have **variable session IDs**, a scheme in which the session ID is **changed** for every request.

Unfortunately, this wouldn't give full protection either. An attacker that got access to a session ID could quickly present it to the web site **before** the victim do a new request.

The attacker would thus completely take over the session, blocking the victim from further access.

The number one measure against session hijacking is to make sure session IDs won't be leaked to third parties in the first place.

Without a valid session ID, session hijacking is very hard.

HTTPS

HTTPS: ENCRYPTED COMMUNICATION

To make a web server secure, **encryption** plays an important role.

In a web setting, encryption usually means **HTTPS**.

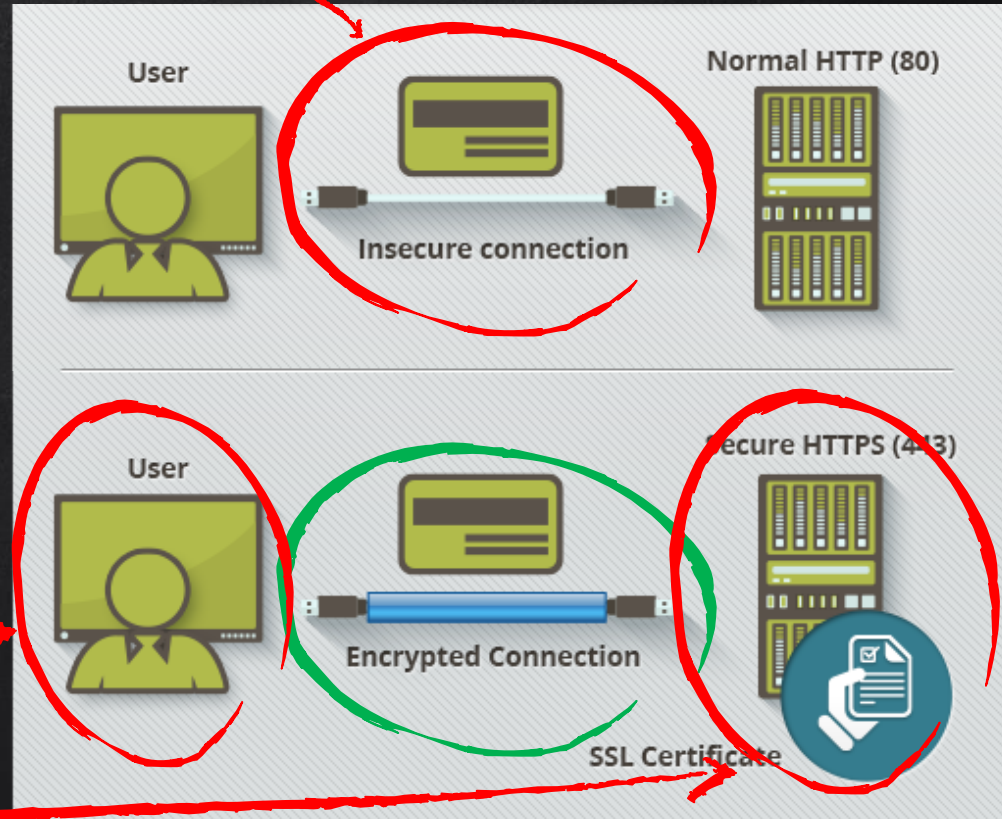


- HTTPS may be described as HTTP communicated over an **encrypted channel**.
- The encrypted channel can be provided by the following protocols:
 - Secure Socket Layer (SSL)
 - Transport Layer Security (TLS)



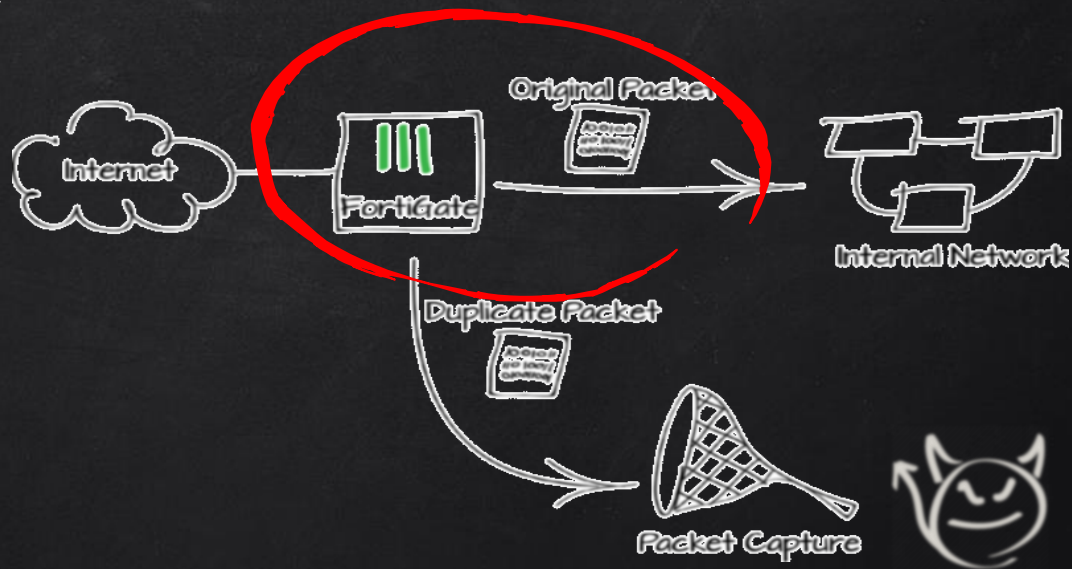
Encryption **only** protects the **network connection** between the client and the server.

An attacker may still attack **both the server and the client**, but he will have a **hard time** attacking the **communication** channel between them.



PACKET SNIFFING

Packet sniffing attacks the **network transport** rather than the application or the client.



The correct approach to protect against sniffing is to **encrypt** all communication.

DEFEAT PACKET SNIFFING USING HTTPS

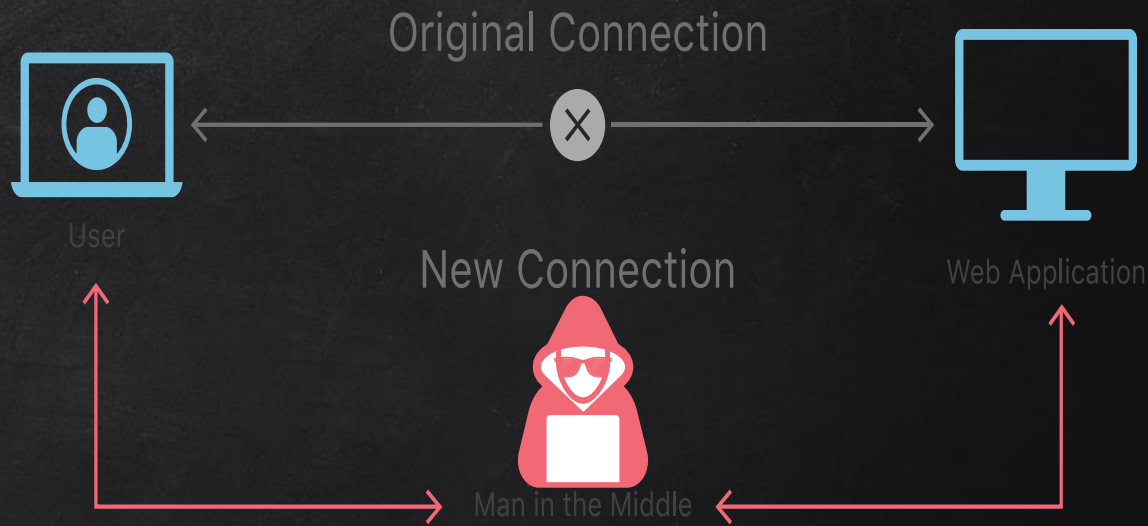
HTTPS makes it impossible for someone to listen to traffic in order to **extract secrets**.

People **may** still sniff packets, but the packets contain seemingly **random data**.



MAN IN THE MIDDLE (MITM)

In MITM, the attacker fools the victim's computer into **connecting to him** rather than the real server (e.g. a bank).



The attacker then **connects to the target server** on behalf of the victim, and effectively sits between the communicating parties, passing messages back and forth.

- He may thus both **listen** to and **modify** the communication.

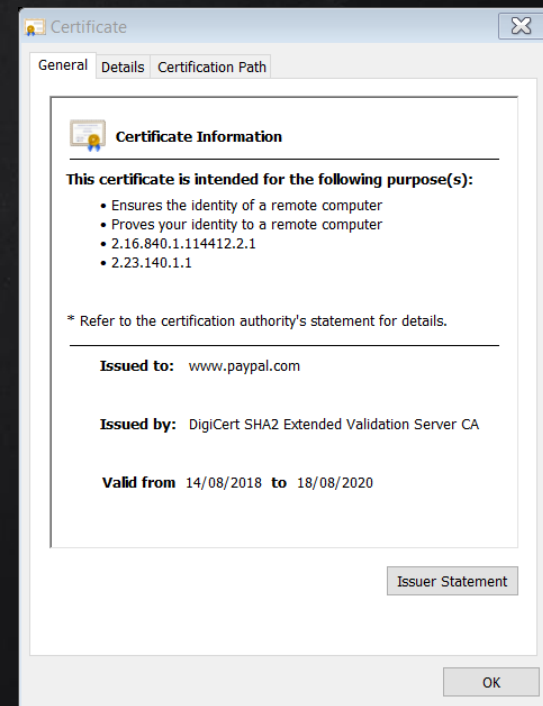


When HTTPS is used, the clients will always verify the **server's certificate**.

Due to the way certificates are generated, the man in the middle **will not be able** to create a fake, but valid certificate for the web site.

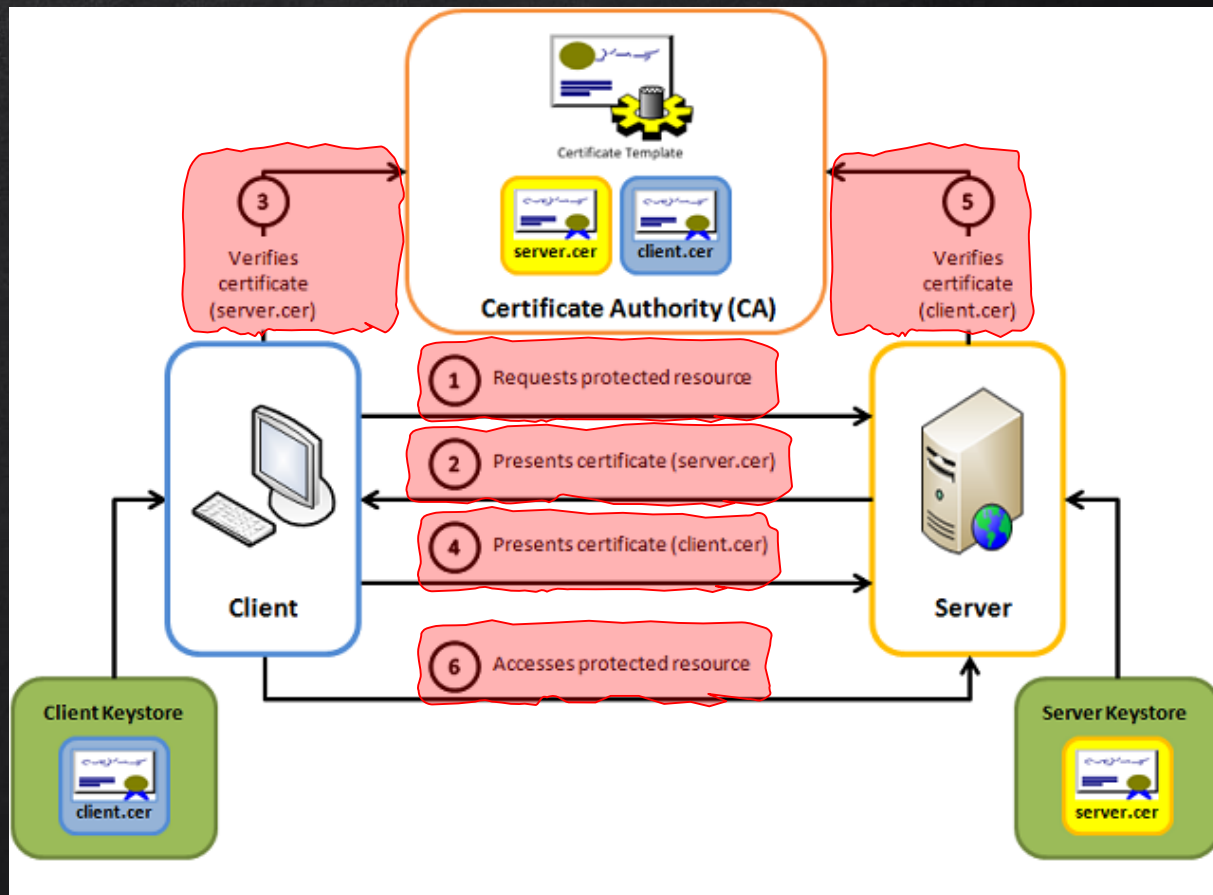
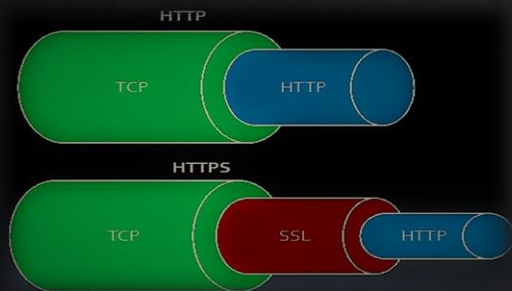
Note that HTTPS in real life doesn't **always** solve the problems completely:

- Ignorant user
- Fake website with similar domain name
- Certification authorities' mistakes
- Bugs in browsers
-



CERTIFICATE BASED MUTUAL AUTHENTICATION

When **HTTPS** is used, the clients will always verify the **server's** certificate.



DEFEAT MITM USING HTTPS

HTTPS provides good protection of the communication channel, unless:

- The user **neglects** the warnings from the browser.
- The **browser** allows the user to neglect its warnings.
- The user **falls** for cheap domain name or protocol **tricks** played by an attacker.
- The **CA** may be tricked into giving out false certificates.
- The **browser** vendor trusts a CA that the user wouldn't trust.
- The browser (or server) has a **buggy SSL/TLS implementation**.
- The user's computer is **controlled** by an attacker.

HTTPS **is not a magic** bullet, however it is a good widely available mechanism for securing web traffic.

SUMMARY

- HTTP is a simple, **text-oriented** protocol.
- **Clients connect** to servers and **send** requests, each of which **draws a response** from the server.
- **Headers** are used to **control** the communication, and to **pass** cookies.
- All request headers and data are **controlled by the client**.
- An attacker **may** thus **easily pass** headers and data after his own liking.

- Sessions are **server-side** containers for **state** information.
- Attackers may be able to **hijack** the sessions of other users by getting hold of their **session ID**.
- Several measures are available to make session hijacking hard, but the real solution is to **keep the session ID a secret**.
- **HTTPS** **protects** data that pass between the client and the server.
- Unfortunately, real world implementations of **HTTPS** **are not** bulletproof. Many weak links play a part in the game, and some of those links may easily break.

YOUR TASKS FOR THIS WEEK



Reading:

- “Innocent Code: A Security Wake-Up Call for Web Programmers (Chapter 1).



Quiz:

- WAS week 1 on the course page on Edmodo.

Note that Quizzes are accessible every Thursday to Sunday only.

